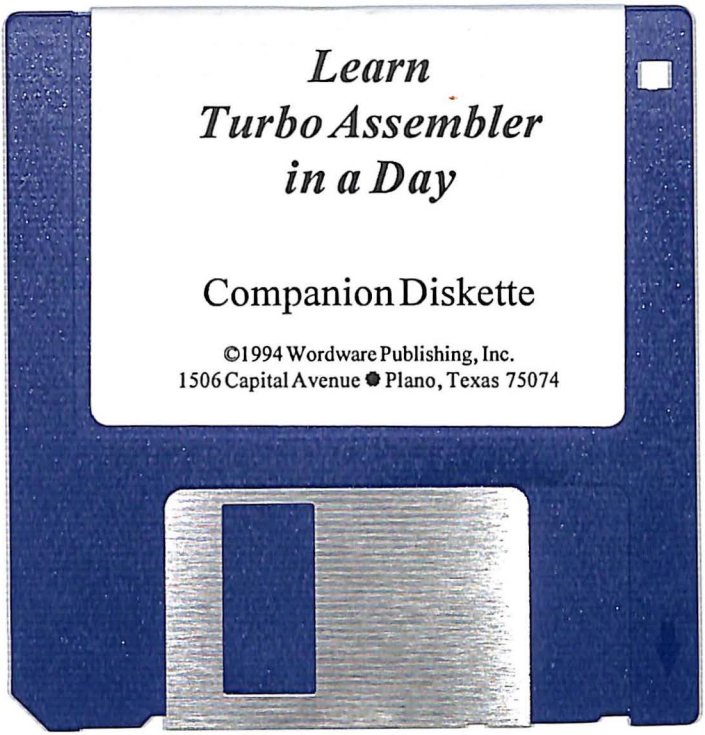
POPULAR APPLICATIONS SERIES

FREE Companion Disk Included

LEARN Turbo Assembler IN A DAY

STEPHEN K. CUNNINGHAM


Learn
Turbo Assembler
in a Day
Companion Diskette
©1994 Wordware Publishing, Inc.
1506 Capital Avenue ● Plano, Texas 75074

Learn Turbo Assembler Programming in a Day

Stephen K. Cunningham

Wordware Publishing, Inc.

Library of Congress Cataloging-in-Publication Data

Cunningham, Stephen K.
Learn Turbo assembler programming in a day / by Stephan K. Cunningham.
p. cm.
Includes index.
ISBN 1-55622-300-5
1. Assembler language (Computer program language) 2. Turbo Assembler (Computer program) I. Title.
QA76.73.A8C86 1992
005.265--dc20 92-24952
CIP

Copyright © 1993, Wordware Publishing, Inc.

All Rights Reserved

1506 Capital Avenue
Plano, Texas 75074

No part of this book may be reproduced in any form or by any means without permission in writing from Wordware Publishing, Inc.

Printed in the United States of America

ISBN 1-55622-300-5
10 9 8 7 6 5 4 3 2 1
9207

PC-DOS is a registered trademark of International Business Machines Corporation.
MS-DOS and Xenix are registered trademarks of Microsoft Corporation.
Other product names mentioned are used for identification purposes only and may be trademarks of their respective companies.

All inquiries for volume purchases of this book should be addressed to Wordware Publishing, Inc., at the above address. Telephone inquiries may be made by calling:

(214) 423-0090

Contents

Introduction

WHO THIS BOOK IS FOR This book is for anyone who wants to learn the basics of programming a PC in its native tongue, which is the 80X86 Assembly language. It discusses practical working code for dealing with specifics of the PC DOS software environment using the Borland Turbo Assembler 3.0. This book can be very useful as a guide between your Turbo Assembler manual, a DOS programmer's manual, and a PC technical reference manual.

WHY READ THIS BOOK People learn to program in PC Assembly language for a variety of reasons:

- to write the fastest possible program routines
- to write the most compact program routines
- to deal directly with hardware when timing cycles count
- to speed up the bottlenecks created by high level compilers
- to learn the basics of the Intel 80X86 instruction set

Assembly language programming is very flexible at solving problems in many different ways. You are not bound by many rigid constraints for coding methodology and data structures with Assembly language programming. Any programming task that can be done in a high level language can also be done in Assembly language since all high level languages have to compile source code down to Assembly language code level for CPU execution. Knowing the Assembly language of a system helps you to understand the constraints and problems of high level language compilers on the system. The Assembly language is where the software meets the hardware.

WHAT THIS BOOK IS ABOUT This book explains and demonstrates how to use the 80X86 instruction set to program a PC running under DOS. The theme of this book is the use of

the 80X86 Assembly language; therefore, many sections are written in the language. You cannot very easily learn to write until you know how to read.

The book begins with a review of the basic concepts of programming in the 80X86 Assembly language. Then the book consists of working code examples with comments about what the code does. The examples demonstrate the following: how to open, read, and write to files; how to write characters to video screens; how to write to the printer; how to use the system clock; and how to use the PC sound port. These code examples are also included on the companion diskette. The book then discusses the use of video graphics, computer math, and C language interface.

There are less than a hundred different mnemonics in the standard 80X86 instruction set; these are not very hard to memorize or quickly reference from a manual. What is more complicated is that many instructions have dozens of different addressing modes. Fortunately, most program routines only need a few of these instructions. Because of this, the book does not explain the complete functional logic behind each and every instruction but, after reading this book, the user will be better able to read and understand an official Intel 80X86 programmer's reference guide.

This book displays examples of good modular code where it is possible. There are sections where you may find routines with more than one entry point and more than one exit point. In these cases, it is done for either code speed reasons or compact code reasons.

Working code examples are included on the companion diskette. See the readme file on the companion diskette for more information.

REQUIREMENTS Hardware: A standard PC with 512K RAM. Software: Turbo Assembler 3.0 or higher and DOS 2.1 or higher. User: Should be familiar with how to use DOS on a PC and know one computer programming language.

Section 1

BEFORE STARTING

INTRODUCTION This section quickly covers some of the elementary concepts about data, the CPU, and general programming. An experienced programmer may want to skip this section.

BITS AND BYTES When programming in Assembly language, it is necessary to know about bits, bytes, and words. A bit is the smallest element of information that can be addressed. A single bit can only be programmed as either zero or one; therefore, it cannot convey much information. A group of eight bits linked together make one byte. With eight bits linked together, you can represent the numbers from zero to 255 or 256 combinations. If the high bit of a byte is used to identify a number as positive or negative, then you can represent the numbers –128 to +127. When the high bit of the byte is used as a positive/negative flag, then the number is said to be signed. This is noted because some instructions will use either signed or unsigned logic. Two 8 bit bytes are linked together to make one 16 bit word. For 16 bits, you can represent the numbers from zero to 65535. The 8086 is classified as a 16 bit processor. Most of the registers are 16 bit with some registers that can also be addressed in 8 bit modes. Most of the operations are word or byte oriented. A signed word can range from –32768 to +32767. A double word is 32 bits which is equal to two words or four bytes.

```
BIT     |X|          (X can be either 0 or 1)
BYTE    |X|X|X|X|X|X|X|X|
WORD    |X|X|X|X|X|X|X|X|X|X|X|X|X|X|X|X|
```

CPU BASICS The CPU is the Central Processing Unit. It executes functions out of a block of memory that can be called its addressing range. It has an instruction pointer that is used to index the next instruction for execution. When the CPU starts executing an instruction, it loads the instruction data into its internal registers while incrementing the instruction pointer to index the next instruction to execute after it finishes execution of the current instruction. Note that some instructions reset the contents of the instruction pointer register to force program execution to alter from the standard next instruction path. This is called program jumping or branching. The 8086 CPU normally operates on data in bytes and words. The 8086 CPU has a 20 bit wide addressing bus which allows it to address over one million bytes of direct memory.

BASICS ABOUT PROGRAMMING The general outline for a standard program routine is one that:

1. reads in data from a device (such as a file, keyboard, memory variable, clock device, mouse device, etc.),
2. then examines or modifies the data if necessary,
3. then writes data out to a device (such as a file, video display, memory variable, sound port, etc.).

An Assembly language program is a list of instruction statements in a source code file that can be compiled into an executable code file that the CPU can understand. Once you have generated the executable code file, you should be able to load the code file into the computer's memory and execute it. In Assembly language programming, many of the statements you write are concerned with controlling the activities of the CPU. You also write directives for compiler activity and comments about the purpose of the procedures. Your program may interface with other devices such as math coprocessors, interrupt controllers, clocking devices, etc. The 8086 CPU has many registers that it uses to perform different tasks. There

are 14 registers available for the programmer to control. Every instruction that is executed in a program affects one or more CPU registers in some way. Even the no operation instruction changes the instruction pointer register. Some registers can be used to hold data. Other registers can be used to index data. Because of the limited number of registers, you may find indexing registers holding data at times.

Section 2

CPU REGISTERS

INTRODUCTION This section will cover the programmable CPU registers. The Assembly language programmer must be familiar with the references and uses of each register in order to program effectively. Other registers are discussed in the section on different 80X86 processors.

The following is a list of the 16 bit registers and their references:

AX Accumulator
FL Flags
X|X|X|X|OF|DF|IF|TF|SF|ZF|X|AF|X|PF|X|CF
BX Base Index
BP Base Pointer
CX Counter
DX Data I/O Index
DI Destination Index
SI Source Index
SP Stack Pointer
IP Instruction Pointer
CS Code Segment
SS Stack Segment
DS Data Segment
ES Extra Segment

Some registers can be divided and accessed as two eight bit registers for byte operations. They are referenced as follows:

AH AX High byte
AL AX Low byte, eight bit accumulator
BH BX High byte
BL BX Low byte
CH CX High byte
CL CX Low byte, shift counter
DH DX High byte
DL DX Low byte

AX and AL are the Accumulator

AX = AH+AL. AX is the 16 bit reference for the accumulator; AL is the 8 bit reference. AL is the low half of AX with AH as the high half. The accumulator is the primary data register. Most instructions for handling data execute faster if the data is in the accumulator.

FL Flag Register

The Flag register is a 16 bit data register used to keep track of CPU activity. This includes all logical, arithmetic, and comparing results as well as interrupt controls, debug tracing, string direction flags, etc.

Most conditional jumping instructions use the contents of this register to determine if branch conditions are true or not true.

Flag Bits

X|X|X|X|OF|DF|IF|TF|SF|ZF|X|AF|X|PF|X|CF

The bit positions with X are not defined for the 8086/8088 CPU but are reserved by Intel for use with other processors in the series.

OF Overflow Flag - This bit is set if the last data manipulation caused the high bit to change.

DF Direction Flag - This bit is used by the CPU to decide the direction of string operations. Clearing the bit causes string operations to go forward and setting the bit causes string operations to go backward.

IF Interrupt Flag - This bit can be set or cleared by the programmer to prevent or allow maskable interrupts to occur.

TF Trace Flag - This bit is used in the debugging mode for single stepping through program logic.

SF Sign Flag - This bit is reset by logical operations to be equal to the high bit of the resulting data.

ZF Zero Flag - This bit is set if the last data manipulation produced a zero condition.

AF Aux Carry Flag - This bit is used by logical instructions that deal with data in nibbles (four bit).

PF Parity Flag - This bit is reset by the last data manipulation instruction to reflect if the operation produced an even or odd parity condition. A 1 means even parity and a 0 means odd parity.

CF Carry Flag - When adding, the carry bit is set if an overflow occurs. If subtracting, the bit is set if it had to borrow a bit because the subtraction resulted in a sign flip.

BX Base Index

BX is the most flexible of the indexing registers. It is a 16 bit register that can also be addressed in eight bit format as BH (high) and BL (low) where BX = BH+BL. BX may be added to other index registers for working with more complex indexing offsets. Examples: [BX+offset], [BX+SI+offset], [BX+DI+offset]

BP Base Pointer

BP is the base pointer register used to index data in the stack area. This register is used by many compilers to index data frames in the stack area. BP may be combined with DI or SI to index data in the stack area. Examples: [BP+SI+offset], [BP+DI+offset]

SI Source Index

SI is the source index register used by the string instructions. It may be combined with BX or BP to index data. Example: [SI+BX+offset]

DI Destination Index

DI is the destination index register used by the string instructions. It may be combined with BX or BP to index data. Example: [DI+BX+offset]

SP Stack Pointer

SP is the stack pointing register used by the push, pop, call, interrupt, and return instructions. It always indexes the last word pushed onto the stack.

CX Counter

CX is the counter register used by the string, repeat, and loop instructions.

DX Data Register, I/O index

DX is the data register. DX is only used as an index for I/O port functions. It is used for the 16 by 16 bit multiply and the 32 by 16 bit divide instructions. The results from a 16 bit multiply are put into DX:AX where DX holds the high 16 bits and AX holds the low 16 bits. For a 32 bit by 16 bit divide, DX will hold the leftover (modulo) data resulting from the divide.

IP Instruction Pointer

IP is used to index the next instruction to execute. It is reset by call and jump instructions.

SEGMENT ADDRESSING REGISTERS The 8086 CPU divides its memory addressing into four areas. The four areas are code, stack, data, and extra data. The current location of these sections is controlled by four segment addressing registers. The 8086 architecture uses these to expand the addressing range of the CPU. The basic addressing range of a normal 16 bit CPU is 65536 bytes. By adding segmented memory offsets into the memory addressing hardware, the addressing range of the 8086 CPU is increased to 20 bits or 1,048,576 bytes. This is done by shifting the four 16 bit segment registers over a nibble (four bits) and adding them to the other index registers to complete the 20 bit real address. With this system, the CPU can address one megabyte of memory. The only complexity to the memory system is that it is divided

into four blocks that have a maximum of 64KB each. This limits the active addressing range of the CPU to 256KB maximum at one time. Because there are two 16 bit words used to complete an address, this book uses "offset" to refer to the address in the lower 16 bit range (0 - 65535). References to the segment address part which is the upper 16 bit word use "SEG" or "segment."

CS Code Segment

This is used with the IP (instruction pointer) register to index the next instruction for program logic execution.

How the CPU calculates a code address with the instruction pointer:

```
 IP                |X|X|X|X|X|X|X|X|X|X|X|X|X|X|X|X|
+ CS          |X|X|X|X|X|X|X|X|X|X|X|X|X|X|X|X|
= real address |X|X|X|X|X|X|X|X|X|X|X|X|X|X|X|X|X|X|X|X|
```

If IP=7 and CS=3 then real address=37H as shown:

```
 IP                |0|0|0|0|0|0|0|0|0|0|0|0|0|1|1|1|
+ CS          |0|0|0|0|0|0|0|0|0|0|0|0|0|1|1|
= real address |0|0|0|0|0|0|0|0|0|0|0|0|0|1|1|0|1|1|1|
```

DS Data Segment

This is used as the primary data area. It is indexed by BX, SI, DI (when DI is not executing string instructions), and offsets without an index register.

SS Stack Segment

This is used as the stacking data area. It is indexed by SP and BP. Note that when BX is used with BP in calculating an address offset, then the segment used is the SS.

ES Extra Segment

This is used as the extra data area. It is indexed by DI during the execution of string instructions as the destination address.

Overrides can be used with most instructions to force an index register to reference data with a different segment register than what is normally used. For example, data indexed by BX normally comes from the DS segment, but with an override, data can come from the CS segment as CS:[BX].

Section 3

THE ELEMENTS OF AN ASSEMBLY LANGUAGE STATEMENT

INTRODUCTION This section is an overview of the basic structure of an Assembly language statement.

There will be many coding examples throughout this book. The coding examples will all be distinguishable by their typewitten style typeface.

In a standard Assembly language statement, there are four fields. The fields are normally separated by a space or a tab character. In many statements, there are fields missing. They can be generally represented and studied from the following example:

Label Field	Operator Field	Data Field	Comment Field

```
;if AH is not zero, this routine adds 30H to AL return
Add_AL_30       PROC    near
        cmp     ah,0            ;compare AH to zero
        jnz     add_al_0        ;jump if not zero, else
        add     al,30H          ;this statement adds 30 to AL
add_al_0:
    ;return to caller
        ret
```

Many program listings start with comments by the program that usually explain the purpose of the code along with additional information. The comment field always starts with the semicolon (;). If the other three fields are empty, then the comment field can begin at the start of a line. The comment field can never come before any other fields on the line. If the comment is more than one line long, then you need a new semicolon at the start of the comment on each line. All comments are optional and are usually put there by the programmer to keep track of program logic.

The label field when used always comes before any other field on a line. This field is used to define names that the programmer creates to reference that location in the program. Here are three ways to define a label:

1. terminating the label with the colon character (:)
2. using the PROC statement if the label is for a coding procedure
3. using the LABEL statement if the label is for a data structure or a data type (NEAR, FAR, BYTE, WORD, DWORD, FWORD, PWORD, QWORD, TBYTE, DATAPRT, CODEPTR)

The operator field comes after the label field and before the data field. This directs the compiler or the CPU to do something. This field is often filled in with language mnemonics.

The data field comes after the operator field and before the comment field. If the operator field requires variable data, then the data goes here. There may be from none to many data variables or operands for an operation. When two data variables are used with most common 8086 instructions, the first operand is the destination and the second operand is the source.

Examples:

```
nop                     ;this has no data variables
jmp     somewhere       ;this has 1 data variable
add     ax,bx           ;this has 2 data variables
macro   1,2,3,4,5       ;a macro may have many.
```

Throughout the examples in this book, there is a wide range of coding styles. This is done to demonstrate some of the different ways to write Assembly language code and document it.

Section 4

COMPILING AND EXECUTING PROGRAMS

INTRODUCTION This section discusses the basics of generating an executable file from an Assembly language source code file. There is also a discussion of the compiler options of conditional assembly and macros.

There are two basic steps used to generate an executable file to run under DOS. The first step is to compile the Assembly language source code into an object file. The second step is to link the object file with any other object files necessary and any needed library routines to generate the executable file. You then type in the name of the executable file at the DOS command line prompt to execute the program.

The following examples use the Borland Turbo Assembler (TASM) for compiling and linking operations. When using TASM to compile Assembly language code, there are four default filename extensions used for the files. These are .ASM for the Assembly language file, .OBJ for the object files, .LIB for the library files, and .EXE for the executable file.

COMPILING Instructions for compiling a program are specific to the compiling software product and there may be many different options. A simple way to compile is to enter the following at the DOS prompt:

```
TASM filename
```

This command performs a standard compile of the file with a .ASM extension and produces a file with the same name but with a .OBJ extension. An example of a command line with a compiler option is as follows:

TASM filename /Z

The /Z is used to help find program syntax errors at compile time. The compiling phase is where you find most of the spelling and syntax errors.

LINKING Instructions for linking a program are specific to the linking software product and there are many different options available. An example of a simple command line link is:

TLINK filename

This will work for some small programs, but with most larger programs you will need to link several small program modules into one large execution file. There are two basic ways to link modules: one way is direct linking and the other way is to link with the library manager to a set of routines in a library file. Linking will catch "unresolved extrn" errors.

Section 5

CODE EXAMPLE: SIMPLE IO PROGRAM

INTRODUCTION The following example illustrates a simple program that gets keyboard input from a user, adds two numbers together, and displays output to the standard video output. This routine uses three basic DOS calls: the DOS function 1 (StdConInput) for input, the DOS function 2 (StdConOutput) for output, and the DOS function 4CH (Exit) to terminate the routine. This routine gets two single digit numbers from a user, then adds the numbers together and displays the results. The routine continues to operate until the user enters a keyboard character that is not a number.

```
;Code routine to add numbers
.MODEL small
;#####################################
.STACK 500
;#####################################
.DATA
data_1  dw      0
;#####################################
.CODE
start   proc    near
        mov     dx,@data        ;get data segment index
        mov     ds,dx           ;set data segment
add_loop:
        call    get_number
        jc      number_error
```

```
        mov     data_1,ax
        call    get_number
        jc      number_error
        add     ax,data_1
        call    display_number
        jmp     add_loop
;!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
number_error:
        mov     ah,4CH          ;set for DOS terminate function
        mov     al,0            ;set terminating variable code
        int     21H             ;call DOS to terminate
start   endp
;************************************
get_number proc near
;on exit if carry clear then number in AX is OK else ERROR
        call    display_new_line
        mov     ah,1    ;set for read keyboard function
        int     21H     ;call DOS to get keyboard data
        cmp     al,'0'  ;test for valid number
        jb      get_number_bad  ;jump if bad number
        cmp     al,'9'
        ja      get_number_bad  ;jump if bad number
        and     ax,0FH
        clc             ;set OK exit code
        ret             ;exit
get_number_bad:
        stc             ;set ERROR exit code
        ret             ;exit subroutine
get_number endp
;************************************
display_number proc near
;on exit binary number in AX is displayed (between 0 to 19)
        call    display_new_line
display_number_1:
        cmp     al,9
        ja      display_number_2
        or      al,30H
        mov     dl,al
        mov     ah,2    ;set for DOS display character
        int     21H     ;call DOS function
        ret             ;exit subroutine
display_number_2:
        sub     al,10   ;adjust number
        push    ax      ;save adjusted number
        mov     dl,'1'
```

```
        mov     ah,2    ;set for DOS display character
        int     21H     ;call DOS function
        pop     ax      ;restore adjusted number
        jmp     display_number_1
display_number endp
;************************************
display_new_line proc near
;on exit cursor is at start of next line down
        push    ax      ;save AX register
        push    dx      ;save DX register
        mov     dl,0DH  ;start of line character
        mov     ah,2    ;set for DOS display character
        int     21H     ;call DOS function
        mov     dl,0AH  ;new line character
        mov     ah,2    ;set for DOS display character
        int     21H     ;call DOS function
        pop     dx      ;restore DX register
        pop     ax      ;restore AX register
        ret             ;exit subroutine
display_new_line endp
;************************************
        end     start
```

Section 6

CODE EXAMPLE: PRINT FILE PROGRAM

INTRODUCTION The following example is a simple program to read a file and print the contents to a standard printer. It gets the filename of the file to print from the DOS command prompt input line. The prompt input information is passed to the program in a buffer area of the Program Segment Prefix (PSP). The address of the Program Segment Prefix is passed to the program in the ES and DS registers when the program execution starts. This program checks the keyboard between every character printed for an escape key code to terminate execution of the program.

```
; This program prints a file defined on the command line
.MODEL small
;*************  Stack Section  *********************
.STACK 500
;*************  Data Section  ***********************
.DATA
psp_seg            dw    0
no_cl_mess  db  "This routine requires that a "
        db     "filename be on the command line for printing."
        db 0dh,0ah,"Please try with a filename.",0dh,0ah,"$"
file_bad_open      db    "Bad file open",0dh,0ah,"$"
file_bad_read      db    "Bad file read",0dh,0ah,"$"
printer_bad_mess   db    "!! Printer Error !!!!",0dh,0ah,"$"
printing_mess      db    "A file is being printed,",0dh,0ah
        db     "To stop printing, Press ESC key",0dh,0ah,"$"
```

```
filename        db      128 dup(0)
file_handle     dw      0
file_count      dw      0
file_pointer    dw      offset file_buffer
file_buffer     db      1024 dup(0)
; ************* ----------- *********************
;*************  Code Section  *******************
.CODE
start   proc    near
   ;DS and ES are indexing PSP area
        mov     al,[DS:80H]       ;load AL with size of data line
        mov     dx,@data          ;get segment address of data area
        mov     ds,dx             ;point DS to data area
        mov     psp_seg,ES        ;save PSP address
        cmp     al,1              ;?? data in DOS command line ??
        ja      get_PSP_filename  ;branch if data found
  ;if here, there is no data on command line
     ;display error message to user and terminate
        lea     dx,no_cl_mess
;-------------------------
terminate_display:
    ;display message indexed by DX then terminate
        mov     ah,09
        int     21H             ;DOS Call
;-------------------------
terminate_program:
;terminating the program
        mov     ah,4CH  ;set AH for terminating function
        mov     al,00   ;set terminating code variable
        int     21H     ;call DOS to terminate
;-------------------------
; %%%%%%%%%%%%%% ----------- %%%%%%%%%%%%%%%%
get_PSP_filename:
  ;move PSP filename to filename buffer in our data area
        mov     ax,ds
        mov     es,ax           ;point ES to data segment
        mov     ds,psp_seg
        mov     si,82H          ;SI source is PSP data area
        lea     di,filename
        cld                     ;make strings go forward
    get_PSP_data_1:
                lodsb           ;load string data byte
               ;check for end of filename
                cmp     al,21H
               ;branch if end of string
```

```
                jb      got_PSP_filename
                stosb           ;store string data byte
                jmp     get_PSP_data_1
got_PSP_filename:
        mov     al,0
        stosb           ;make ASCIIZ string with zero end
        push    es
        pop     ds      ;reset data segment pointer
;try to open file
        mov     ah,3dH
        lea     dx,filename
        mov     al,0            ;read access code
        int     21H             ;DOS Call
        jnc     file_open_ok
        lea     dx,file_bad_open
        jmp     terminate_display
;++++++++++++++++++++++++++++++++++++++++++++
;############### ++++++++++++ ###############
file_open_ok:
    ;save file handle
        mov     file_handle,ax
        lea     dx,printing_mess   ;display start message
        mov     ah,09
        int     21H                ;DOS Call

file_read:
   ;read in block of file data
        mov     ah,3fH
        lea     dx,file_buffer
        mov     cx,1024
        mov     bx,file_handle
        int     21H             ;DOS Call
        jnc     file_read_ok    ;branch if good read
            ;else read file error occurred
              ;close file
                mov     ah,3eh
                mov     bx,file_handle
                int     21H
             ;index exit error message
                lea     dx,file_bad_read
                jmp     terminate_display
file_read_ok:
     ;check to see if no more file data
        cmp     ax,0
        je      close_file      ;branch if no data left
```

```
      ;else reset data block size and pointer
        mov     file_count,ax
        lea     bx,file_buffer
        mov     file_pointer,bx
;!!!!!!!!!!!!!!!!!  ^^^^^^^^  !!!!!!!!!!!!!!!!!!!
print_data_block:
   ;main loop to print block of file data
     ;scan keyboard to check for any keys
        mov     ah,1
        int     16H
        jz      print_data_block_1      ;branch if no key
      ;get key code out of buffer
        mov     ah,0
        int     16H             ;call BIOS keyboard
        cmp     al,01BH         ;check key code
        je      close_file      ;branch if ESC
print_data_block_1:
        mov     si,file_pointer
        mov     al,[si]
        mov     ah,0
        mov     dx,0            ;select LPT1
        int     17H             ;BIOS Call
        test    ah,25H
        jnz     printer_error
        inc     si
        mov     file_pointer,si
        dec     file_count
        jnz     print_data_block        ;loop if more data
    ;else go read in next block of file data
        jmp     file_read
;!!!!!!!!!!!!!!!!  ^^^^^^^^  !!!!!!!!!!!!!!!!!!!
close_file:
        mov     ah,3eh
        mov     bx,file_handle
        int     21H             ;DOS Call
        jmp     terminate_program
;--------------  ??????????  --------------
printer_error:
             ;index exit error message
                lea     dx,printer_bad_mess
                jmp     terminate_display
;_____________________________________________________
start   endp            ;end of start procedure
        end     start   ;define start as beginning of program
```

Section 7

CODE EXAMPLE: GET TIME OF DAY

INTRODUCTION The first example is of a program that uses the system time of day. This example uses a conditional assembly to decide between two ways to get the time of day. One way is to use the DOS time of day function or the other way, illustrated by the second code example, uses the BIOS time of day function. The second example can be used to replace the standard DOS call to get the time of day. This second example will return the time of day much faster than the standard DOS call.

FIRST EXAMPLE This example runs until a set time of day occurs and then terminates. This code gets the time of day information from the DOS command line input data. If there is bad input data, then an error message is displayed, informing the user what the correct input data format is. While the program is waiting for the terminate time, it will scan the keyboard for an escape key to terminate the program on user demand.

```
; Wait for Time
;This program can be executed inside of a .BAT file to
;  stall execution of the .BAT file until a set time of day
;????????????????????????????????????????????????????????????
;conditional assembly flag, 0 to use DOS, 1 to use BIOS example
.MODEL tiny
```

```
use_bios_flag EQU 1
;----------- stack area -----------
.STACK 500
;--------------------------------
.CODE
;+++ this routine combines data and code into one segment +++
; define data area
psp_seg         dw      0
wait_hour       db      0
wait_minute     db      0

wait_message    db     0DH,0AH,0DH,0AH
        db   "Wait in progress, Press [ESC] to exit",0DH,0AH
                db      "$"
exit_bad_message        db      0DH,0AH
        db   "To use TimeWait program enter timeout data "
        db   "from command line as example:",0DH,0AH,0DH,0AH
        db     "TimeWait 11:30",0DH,0AH,0DH,0AH
        db     "Note, timeout hours vary from 0 to 23, "
        db      "and minutes from 0 to 59.",0DH,0AH
                db      "$"

;*************  @@@@@@@@@@@@@@  ***************
;start code area
start   proc    near
        mov     bx,80H          ;index command line data
        mov     al,[bx]         ;get size of string variable
        mov     ax,cs
        mov     ds,ax           ;reset data segment
        mov     psp_seg,es      ;save PSP address
        mov     es,ax           ;reset extra segment
        cmp     al,4            ;is there data in string
        jb      exit_bad        ;branch if no data
        inc     bx
        inc     bx              ;point to start of data
      ;get number out of buffer area
        call    get_number
        jc      exit_bad        ;branch if number bad
        mov     wait_hour,al    ;save number in hour
        cmp     al,23           ;?? number too large ??
        ja      exit_bad        ;branch is too large
      ;check the number terminating character
        cmp     ah,":"
        jne     exit_bad        ;branch if not :
      ;point to start of next number
```

```
        inc     bx
      ;get next number out of buffer area
        call    get_number
        jc      exit_bad        ;branch if number bad
        cmp     al,59           ;?? number too large ??
        ja      exit_bad        ;branch if too large
        mov     wait_minute,al  ;save number to minute
   ;display executing wait message
        mov     ah,9        ;set DOS function number
        lea     dx,wait_message
        int     21H         ;DOS call to display message
;_____________________________________
;**********  !!!!!!!!  **********
wait_loop:
     ;scan keyboard for keys
        mov     ah,1
        int     16H
        jz      wait_no_key     ;branch if no key
        mov     ah,0            ;if here then keyboard data
        int     16H             ;get key code from buffer
        cmp     ax,3B00H        ;check key code
        je      exit            ;branch if exit key
        cmp     al,1BH          ;check for ESC key
        je      exit            ;branch if ESC key
wait_no_key:
   ;find out what time it is
;conditional assembly ??????????????????????
     ;use this code if linking to code in this section
IF use_bios_flag
        call    get_time_of_day
     ;else use this code if calling DOS for time
ELSE
        mov     ah,2CH
        int     21H             ;get current time of day
ENDIF
        cmp     ch,wait_hour
        jne     wait_loop       ;loop if not time
        cmp     cl,wait_minute
        jne     wait_loop       ;loop if not time
;^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
exit:
        mov     ah,4CH
        int     21h             ;terminate program
;!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
exit_bad:
```

```
        mov     ah,9
        lea     dx,exit_bad_message
        int     21H             ;DOS call to display message
        jmp     exit
;  ***** ^^^^^^^^ ***** ^^^^^^^^ ****
get_number:
 ;on entry BX indexes ASCII number data in PSP segment area
 ;on exit if carry clear,
  ; register AL has binary number, from 0 to 99
  ; BX indexes past the number,
  ; AH has exiting character code indexed by BX
        push    ds
        mov     ds,psp_seg
        mov     al,[bx]
        inc     bx
        call    number_check
        jc      get_number_bad
        mov     ah,al
        mov     al,[bx]
        call    number_check
        jc      get_number_1
get_number_2a:
        cmp     ah,0
        je      get_number_2
        add     al,10
        dec     ah
        jmp     get_number_2a
get_number_2:
        inc     bx
        mov     ah,al
        mov     al,[bx]
get_number_1:
        cmp     al,":"
        je      get_number_1a
        cmp     al,0DH
        jne     get_number_bad
get_number_1a:
        xchg    al,ah
        pop     ds
        clc             ;set good number flag
        ret
get_number_bad:
        pop     ds
        stc             ;set bad number flag
        ret
```

```
;###############################################
number_check:
   ;this code checks for ASCII number in AL
    ; if it finds a number, then it makes it binary
    ; and returns with carry clear, else carry set
        cmp     al,"0"
        jb      number_bad
        cmp     al,"9"
        ja      number_bad
        and     al,0FH
        clc
        ret
number_bad:
        stc
        ret
;*************************
start   endp
        end     start
```

SECOND EXAMPLE The biggest problem encountered when working with the PC timer is that it ticks at an odd rate of about 18.2 ticks per second. When you call the BIOS for the time of day, you get a 32 bit value that represents the number of ticks that have passed that day. By dividing this number by about 18.2, you get the number of seconds that have passed that day. At first glance, the number 18.2 doesn't seem to divide very nicely. Therefore, I tried multiplying it, first by 60 (to find out how many ticks are in a minute) then by 60 again (to find out how many ticks are in an hour).

18.2 x 60 = 1092
1092 x 60 = 65520

Note that 65520 is very close to 65536 which is equal to hexadecimal 10000. An actual hour is about 65543 ticks, which shows that 18.2 is not 100% accurate.

Using this information, you can find a quick way to get a time of day value out of the system timer when you are not concerned about it being absolutely accurate. This code can be used for applications that want to get the time of day in hours, minutes, and seconds as fast as possible. This code is very fast, but it suffers from a timing flaw. The code will stretch the midnight second for about 12 seconds to correct for accumu-

lated calculation errors. For many applications, this will not be a problem. This code is not designed for applications that need timing values to be as close to the real time of day as possible.

The following example uses the PC BIOS INT 1AH to get the 32 bit time of day count from the system timer. It then takes the count and generates the current time of day in hours, minutes, and seconds. This code puts the data into the same registers as the DOS call returns. The trick to this routine is that it assumes that the top word of the 32 bit value is the hour count and the lower word is the minute/second count. This code generates a value that is very close to the real time of day. If this code is used without the 12 second fix, then a time of day of 24:00:10 will occur. With this midnight fix, the time clock goes up to 23:59:59 and then over to 00:00:00 for about 12 seconds.

```
.CODE
IF use_bios_flag
get_time_of_day proc    near
  ;on exit CH has hours,CL has minutes,DH has seconds,
   ; DL has leftover (modulo)
        public  get_time_of_day
        push    ax
        push    bx
              ;AH to 0 for BIOS get time of day call
                xor     ax,ax
                int     1AH             ;call BIOS
                cmp     cx,24           ;check for midnight
              ;branch if midnight
                jae     get_time_of_day_mn
              ;else compute time of day
                mov     ch,cl           ;put hours in CH
                mov     bx,1092         ;65536/60
                mov     ax,dx
                xor     dx,dx
                div     bx
                mov     cl,al           ;put minutes in CL
                mov     ax,dx
                xor     dx,dx
                mov     bx,18           ;(65536/60)/60
                div     bx
                mov     dh,al           ;put seconds in DH
```

```
        pop     bx
        pop     ax
        ret

get_time_of_day_mn:
    ;fix for 12 second midnight
     ; there are 1800B0h ticks in a day
                xor     cx,cx
                xor     dx,dx

        pop     bx
        pop     ax
        ret
;*********************************
get_time_of_day endp
ENDIF
```

Section 8

CODE EXAMPLE: MAKE SOUNDS

INTRODUCTION This code example provides a set of keyboard routines to control sound output while waiting for a user to enter a keyboard character. The advantage to this method is that a main routine can call these sound routines to play a sound sequence and the sound routines will return control back to the main routine whenever the user enters keyboard data so that the main routine can continue computing while the sound plays in the background. The code example has two different code entry points for getting keyboard data. One code entry point is a standard get_keyinput call which will wait for a key and update the sound data until a key code is found. The other code entry point is the get_keyinput_to call which will wait a set amount of time for a key code and if none is found, return with a no key code found condition. The calling routine puts a timeout counter value in register AX on entry. The counter value is based on the system clock which ticks at 18.2 times per second. The entry point start_table_sound is used to begin a background sound sequence. On entry, the register BX indexes a table of sound data. The table has a format of four byte entries and is terminated by a data word of zero. The four bytes are used as two words: the first is a duration count and the second is a tone value. There are two code entry points for turning the background sound off and on. There is also a

utility to flush out the keyboard buffer that can be executed with a call to flush_keyboard.

```
;Set of keyboard routines with sound outputs
.MODEL small
.STACK 500
.DATA
  ;define table for sound output
sample_sounds           dw      8,45000   ;long low sound
                        dw      2,2000    ;short high sound
                        dw      0         ;end of sample sound table

sound_table     dw      0
sound_time_m    dw      0
sound_time_l    dw      0
sound_flag      db      0
sound_on_flag   db      0,0
key_time_out_m dw       0
key_time_out_l dw       0

.CODE
;************  ^^^^^^^^^^  *************
;### code entry point #####
get_keyinput    proc    near
; this routine checks for keyboard data in BIOS buffer
; and returns with data if there
; else it updates sound output data and loops to check for
; keyboard data again until keyboard data found
; on exit AX has keyboard data
        public  get_keyinput
        push    bx
        push    cx
        push    dx
get_keyinput_loop:
                mov     ah,1    ;set AH for scan
                int     16H     ;BIOS Call
                                ;branch if no keyboard data
                jz      sound_update
                mov     ah,0    ;set AH for get key
                int     16H     ;BIOS Call
        pop     dx
        pop     cx
        pop     bx
        ret
```

```
;*******  -------- *******
sound_update:
        cmp     sound_flag,0            ;check for sound on????
        jz      get_keyinput_loop       ;branch out if sound off
        mov     cx,sound_time_m         ;else check for sound update
        mov     ax,sound_time_l
        call    test_current_time       ;is it time for update ??
        jc      get_keyinput_loop       ;branch if not time
        mov     bx,sound_table
        mov     ax,[bx]                 ;get next sound update value
        or      ax,ax                   ;?? end of sound ??
        jz      turn_sound_off          ;branch if end sound
        call    get_time_plus_ax        ;reset sound duration
        mov     sound_time_m,cx
        mov     sound_time_l,ax
        inc     bx
        inc     bx
        mov     ax,[bx]
        inc     bx
        inc     bx
        mov     sound_table,bx
        call    sound_out_ax            ;go set sound frequency
        jmp     get_keyinput_loop       ;branch to keyboard loop
turn_sound_off:
        call    sound_off
        mov     sound_flag,0
        jmp     get_keyinput_loop       ;branch to keyboard loop
get_keyinput    endp

;^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
;************* ########## **************
;### code entry point #####
get_keyinput_to proc    near
;get keyboard data with timeout if no data available
;on entry AX has time duration in 18 ticks per second
;on exit if carry clear then AX has keyboard data
        public  get_keyinput_to
        push    bx
        push    cx
        push    dx
        call    get_time_plus_ax     ;add duration to current time
        mov     key_time_out_m,cx    ;set timeout value
        mov     key_time_out_l,ax
get_keyinput_to_loop:
        mov     ah,1            ;ready to scan keyboard data
```

```
        int     16H             ;BIOS Call
        jz      sound_update_to ;branch if no keyboard data
        mov     ah,0                    ;ready to get key data
        int     16H                     ;BIOS Call
        pop     dx
        pop     cx
        pop     bx
        clc                             ;set keyboard data flag
        ret
get_keyinput_to_1:
        mov     cx,key_time_out_m       ;check for timeout
        mov     ax,key_time_out_l
        call    test_current_time
        jc      get_keyinput_to_loop    ;branch if no timeout
        xor     ax,ax                   ;else timeout return condition
        pop     dx
        pop     cx
        pop     bx
        stc                             ;set no keyboard data flag
        ret

; ********  %%%%%%%  ********
sound_update_to:
        cmp     sound_flag,0            ;check for sound on????
        jz      get_keyinput_to_1       ;branch if sound off
        mov     cx,sound_time_m         ;else check for sound update
        mov     ax,sound_time_l
        call    test_current_time
        jc      get_keyinput_to_1       ;branch if not ready for update
        mov     bx,sound_table
        mov     ax,[bx]
        or      ax,ax                   ;test for end of table
        jz      turn_sound_off_to       ;branch if end of table data
        call    get_time_plus_ax
        mov     sound_time_m,cx
        mov     sound_time_l,ax
        inc     bx
        inc     bx
        mov     ax,[bx]
        inc     bx
        inc     bx
        mov     sound_table,bx
        call    sound_out_ax
        jmp     get_keyinput_to_1
turn_sound_off_to:
```

```
        call    sound_off
        mov     sound_flag,0
        jmp     get_keyinput_to_1
get_keyinput_to endp
;^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

;*************  @@@@@@@@@@  *************
;### code entry point #####
start_table_sound       proc    near
;subroutine to start background sound output
;on entry BX indexes sound data table
        public  start_table_sound
        push    ax
        push    bx
        mov     ax,[bx]
        call    get_time_plus_ax
        mov     sound_time_m,cx
        mov     sound_time_l,ax
        inc     bx
        inc     bx
        mov     ax,[bx]
        inc     bx
        inc     bx
        mov     sound_table,bx
        call    sound_out_ax
        mov     sound_flag,0FFH
        pop     bx
        pop     ax
        ret
start_table_sound       endp

;*************  ==========  *************
;### code entry point #####
flush_keyboard  proc    near
  ;utility to flush contents of keyboard buffer
        public  flush_keyboard
        mov     ah,1
        int     16H             ;BIOS Call  ;scan for keyboard data
        jz      flush_keyboard_x        ;branch if no keyboard data
        mov     ah,0                    ;else get keyboard data
        int     16H             ;BIOS Call
        jmp     flush_keyboard
flush_keyboard_x:
        ret
flush_keyboard  endp
```

```
;*************  -----------  **************
sound_out_ax    proc    near
;set sound out frequency to data value in AX
        push    ax
        push    ax
        cmp     sound_on_flag,0
        jne     sound_out_1
        in      al,61H          ;input port 61h
        or      al,3
        out     61H,al          ;output port 61h
sound_out_1:
        mov     al,0B6H
        out     43H,al          ;output port 43h
        pop     ax
        out     42H,al          ;output port 42h
        xchg    al,ah
        out     42H,al          ;output port 42h
        mov     sound_on_flag,0FFH
        pop     ax
        ret
sound_out_ax    endp

;************  $$$$$$$$$$  *************
;###### code entry point #######
sound_off       proc    near
   ;turn sound port off
        public  sound_off
        push    ax
        cmp     sound_on_flag,0
        je      sound_off_exit
        in      al,61H          ;input port 61h
        and     al,0FCH
        out     61H,al          ;output port 61h
        mov     sound_on_flag,0
sound_off_exit:
        pop     ax
        ret
sound_off       endp

;***************  %%%%%%%%%%  ***************
;with all CX:AX time values, CX is most significant
; and AX is least significant

get_current_time        proc    near
```

```
;on exit CX:AX has 32 bit day clock value
; in 18.2 ticks per second
        push    dx
                xor     ax,ax   ;set AH to zero
                int     1AH     ;BIOS Call get time
                mov     ax,dx
        pop     dx
        ret
get_current_time        endp

;*****************************
get_time_plus_ax        proc    near
;on entry AX has 16 bit value to add to current clock time
;on exit CX:AX has new 32 bit clock value
        push    dx
        push    ax
        xor     ax,ax
        int     1AH             ;BIOS Call
        pop     ax
        add     ax,dx
        adc     cx,0
        pop     dx
        ret
get_time_plus_ax        endp

;************  ########  ************
test_current_time       proc    near
;on entry CX:AX has time value
; to be subtracted from the current time
;on exit if carry set then current time
; is less than CX:AX time
        push    dx
        push    cx
        push    ax
        xor     ax,ax
        int     1AH             ;BIOS Call
        cmp     dx,18
        jb      test_current_time_2
test_current_time_1:
        pop     ax
        sub     dx,ax
        pop     dx
        sbb     cx,dx
        mov     cx,dx
        pop     dx
```

```
        ret
test_current_time_2:
        or      cx,cx
        jnz     test_current_time_1
        pop     ax      ;this is fix code for midnight factor
        pop     dx
        pop     dx
        clc             ;clear carry condition
        ret
test_current_time       endp

;****************************************
        end
```

Section 9

CODE EXAMPLE: VIDEO CHARACTER INTERFACE

INTRODUCTION This section has a set of code examples for writing character data to a PC in 80 character by 25 line mode. The first routine will check the video mode, and if it is a standard mode, the code returns with a no carry condition and starts using the standard video BIOS routines to perform the requested functions. These routines provide many code entry points where other programs may call in. Before using any of the video routines, the program must call the reset_video routine to initialize and make ready the other video calls. Some of the video display routines will respond to video display escape codes. Most of the escape codes deal with standard cursor control functions. There is an example of table selection jumping code.

Most computers use a raster graphics video system. The data is sent to the monitor as rows from left to right that are arranged from top to bottom. The first data bit sent goes to the upper left corner and the last data bit in a frame goes to the lower right corner. When addressing video RAM, the memory may be divided into rows that can be divided into columns. A fundamental understanding of this system is necessary when translating a row and column position into the actual video RAM address for a character or pixel location.

```
;set of video display routines
.MODEL small
        public  reset_video
        public  clear_screen
        public  get_cursor_position, set_cursor_position
            ;write character in AL, use TTY method
        public  write_to_screen
            ;write string indexed by SI using TTY
        public  write_asciiz_string
            ;write character in AL and display control codes
        public  display_character
        public  save_screen, restore_screen
            ;set by program to activate escape code functions
        public  esc_flag
        public  normal_attribute
        public  scroll_screen_up, scroll_screen_down
        public  screen_buffer, cursor_port
.DATA
;video data variables
        even
cursor                  dw      0
save_cursor             dw      0
cursor_port             dw      3B4H

screen_buffer           dw      2001 dup(0)
esc_flag                        db      0
esc_on_flag             db      0
esc_y_flag              db      0
esc_y_line              db      0
video_hw_mode           db      0
normal_attribute        db      07H

  ;jump table used for ESC codes
esc_jmp_table   db      "A"
                dw      write_esc_a
                db      "B"
                dw      write_esc_b
                db      "C"
                dw      write_esc_c
                db      "D"
                dw      write_esc_d
                db      "H"
                dw      write_esc_h
                db      "I"
                dw      write_esc_i
```

```
                db      "J"
                dw      write_esc_j
                db      "K"
                dw      write_esc_k
                db      "Y"
                dw      write_esc_y
                db      0,0,0

.CODE
;******************  $$$$$$$$$$$$$  ******************
video_routines  proc    near

reset_video:
;this routine needs to be called before any other video
;function routines to start up the video system
;on exit if carry set then video mode less than 80 columns
        push    ax
        push    bx
        mov     ah,15
        int     10H                 ;check current video mode
        cmp     ah,80
        jae     reset_video_80_b    ;branch if 80+ columns
       ;video is less than 80 columns
        stc                         ;set for error condition on exit
        jmp     short reset_video_exit_b
reset_video_80_b:
        mov     video_hw_mode,0FFH
        mov     normal_attribute,07H
        clc
reset_video_exit_b:
        pop     bx
        pop     ax
        ret
;###############################################
;  **************  ############  ****************
clear_screen:
  ;clear screen using BIOS calls
        push    ax
        push    cx
        push    dx
        push    bx
        mov     ax,600H
        mov     cx,0000H
        mov     dh,24
        mov     dl,79
```

```
        mov     bh,normal_attribute
        int     10H
        xor     dx,dx
        call    set_cursor_position
        pop     bx
        pop     dx
        pop     cx
        pop     ax
        ret
;^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
;**************  ^^^^^^^^^^^^  ***************
set_cursor_position:
;set cursor with BIOS call
;on entry have DX set for new cursor position
        push    ax
        push    bx
        push    dx
        mov     ah,2
        mov     cursor,dx
        xor     bx,bx
        int     10H
        pop     dx
        pop     bx
        pop     ax
        ret
;----------------------------------------
;*************  %%%%%%%%%%%%%  *************
get_cursor_position:
;on exit DX has current cursor position
        mov     dx,cursor
        ret
;_____________________________________
;************ ________ *************
write_esc_y_on:
        cmp     esc_y_flag,0FFH
        jne     write_esc_y_on1
        sub     al,20H
        mov     esc_y_line,al
        mov     esc_y_flag,0FH
        jmp     write_to_screen_x
write_esc_y_on1:
        sub     al,20H
        mov     dl,al
        mov     dh,esc_y_line
        cmp     dh,24
```

```
        ja      write_esc_y_er
        cmp     dl,79
        ja      write_esc_y_er
        call    set_cursor_position
write_esc_y_er:
        mov     esc_y_flag,0
        mov     esc_on_flag,0
        jmp     write_to_screen_x
;@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
write_esc_data:
        cmp     esc_y_flag,0
        jne     write_esc_y_on
        lea     bx,esc_jmp_table
write_esc_data_1:
        cmp     al,[bx]
        je      write_esc_data_jmp
        add     bx,3
        cmp     byte ptr[bx],0
        jne     write_esc_data_1
        mov     esc_on_flag,0
        jmp     write_to_screen_x
write_esc_data_jmp:
        mov     bx,[bx+1]
        jmp     bx
;--------------------------------
;***********  %%%%%%  ************
write_to_screen:
;on entry AL has ASCII character for TTY output
; this code uses the system BIOS calls for display functions
        push    ax
        push    bx
        push    cx
        push    dx
        cmp     esc_on_flag,0
        jne     write_esc_data
        mov     bl,normal_attribute
        cmp     al,20H
        jb      write_to_screen_c
write_to_screen_0:
        mov     cx,01
        mov     bh,0
        mov     ah,9
        int     10H
        mov     dx,cursor
        cmp     dl,79
```

```
        je      write_to_screen_1
        inc     dl
        call    set_cursor_position
        jmp     short write_to_screen_x
write_to_screen_1:
        mov     dl,0
        cmp     dh,24
        je      write_to_screen_2
        inc     dh
        call    set_cursor_position
        jmp     short write_to_screen_x
write_to_screen_2:
        call    set_cursor_position
        call    scroll_screen_up
write_to_screen_x:
        pop     dx
        pop     cx
        pop     bx
        pop     ax
        ret
write_to_screen_c:
;  check for special keyboard control codes
        mov     dx,cursor
        cmp     al,0DH
        je      write_to_screen_cr
        cmp     al,0AH
        je      write_to_screen_lf
        cmp     al,09H
        je      write_to_screen_tab
        cmp     al,0CH
        je      write_to_screen_ff
        cmp     al,08H
        je      write_to_screen_bs
        cmp     al,1BH
        je      write_to_screen_esc
        jmp     write_to_screen_0  ;branch if unknown code
write_to_screen_cr:
        mov     dl,0
        call    set_cursor_position
        jmp     write_to_screen_x
write_to_screen_lf:
        cmp     dh,24
        je      write_to_screen_lf1
        inc     dh
        call    set_cursor_position
```

```
        jmp     write_to_screen_x
write_to_screen_lf1:
        call    scroll_screen_up
        jmp     write_to_screen_x
write_to_screen_tab:
        and     dl,0F8H
        add     dl,8
        cmp     dl,80
        je      write_to_screen_tab1
        call    set_cursor_position
        jmp     write_to_screen_x
write_to_screen_tab1:
        mov     dl,0
        cmp     dh,24
        je      write_to_screen_tab2
        inc     dh
        call    set_cursor_position
        jmp     write_to_screen_x
write_to_screen_tab2:
        call    set_cursor_position
        call    scroll_screen_up
        jmp     write_to_screen_x
write_to_screen_ff:
        call    clear_screen
        jmp     write_to_screen_x
write_to_screen_bs:
        or      dl,dl
        jz      write_to_screen_bsx
        dec     dl
        call    set_cursor_position
write_to_screen_bsx:
        jmp     write_to_screen_x
write_to_screen_esc:
        cmp     esc_flag,0
        jne     write_to_screen_esc_1
        mov     esc_on_flag,0FFH
        jmp     write_to_screen_x
write_to_screen_esc_1:
        jmp     write_to_screen_0
;++++++++++++++++++++++++++++++++++++++++++++++
;*************  <<<<<< >>>>>>  **************
  ;the following are for processing escape string functions
write_esc_a:
;move cursor up one line
        mov     dx,cursor
```

```
        cmp     dh,0
        je      write_esc_a_0
        dec     dh
        call    set_cursor_position
write_esc_a_0:
        mov     esc_on_flag,0
        jmp     write_to_screen_x

write_esc_b:
;move cursor down one line
        mov     dx,cursor
        cmp     dh,24
        je      write_esc_b_0
        inc     dh
        call    set_cursor_position
write_esc_b_0:
        mov     esc_on_flag,0
        jmp     write_to_screen_x

write_esc_c:
;move cursor right one character
        mov     dx,cursor
        cmp     dl,79
        je      write_esc_c_0
        inc     dl
        call    set_cursor_position
write_esc_c_0:
        mov     esc_on_flag,0
        jmp     write_to_screen_x

write_esc_d:
;move cursor left one character
        mov     dx,cursor
        cmp     dl,0
        je      write_esc_d_0
        dec     dl
        call    set_cursor_position
write_esc_d_0:
        mov     esc_on_flag,0
        jmp     write_to_screen_x

write_esc_h:
;move cursor to top left position
        xor     dx,dx
        call    set_cursor_position
```

```
        mov     esc_on_flag,0
        jmp     write_to_screen_x

write_esc_i:
;move cursor up with scroll if on top line
        mov     dx,cursor
        cmp     dh,0
        je      write_esc_i_0
        dec     dh
        call    set_cursor_position
        mov     esc_on_flag,0
        jmp     write_to_screen_x
write_esc_i_0:
        call    scroll_screen_down
        mov     esc_on_flag,0
        jmp     write_to_screen_x

write_esc_j:
;erase from cursor to end of screen
        push    cx
        push    bx
        mov     ax,cursor
        mov     cx,79
        sub     cl,al
write_esc_j2:
        cmp     ah,24
        je      write_esc_j1
        add     cx,80
        inc     ah
        jmp     write_esc_j2
write_esc_j1:
        jcxz    write_esc_j3
        mov     bl,normal_attribute
        mov     al,20H
        add     cx,1
        mov     bh,0
        mov     ah,9
        int     10H
write_esc_j3:
        pop     bx
        pop     cx
        mov     esc_on_flag,0
        jmp     write_to_screen_x
```

```
write_esc_k:
;erase from cursor to end of line
        push    cx
        push    bx
        mov     ax,cursor
        mov     cx,79
        sub     cl,al
        jmp     write_esc_j1

write_esc_y:
;set cursor position
        mov     esc_y_flag,0FFH
        jmp     write_to_screen_x
;_____________________________________________
;************  <<<<<<<<< >>>>>>>>>> ************
display_character:
;on entry AL has character for output using BIOS calls
        push    ax
        push    bx
        push    dx
        push    cx
        mov     bl,normal_attribute
        mov     ah,9
        mov     bh,0
        mov     cx,01
        int     10H             ;BIOS call to display
      ;reset cursor position for next character
        mov     dx,cursor
        cmp     dl,79           ;is this end of line ???
       ;branch if end of current line
        je      display_character_1
        inc     dl              ;index next column position
        call    set_cursor_position
       ;go to exit subroutine
        jmp     short display_character_x
display_character_1:
        mov     dl,0            ;index start of line
        cmp     dh,24           ;is this last line ????
       ;branch if last line on screen
        je      display_character_2
        inc     dh              ;index next line
        call    set_cursor_position
       ;go to exit subroutine
        jmp     short display_character_x
display_character_2:
```

```
        call    set_cursor_position
      ;scroll screen up one line for a new line
        call    scroll_screen_up
display_character_x:
  ;exit subroutine
        pop     cx
        pop     dx
        pop     bx
        pop     ax
        ret

;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
;***************  ^^^^^^^^^^  ****************
write_asciiz_string:
;on entry have SI indexing ASCIIZ data
        mov     al,[si]
        or      al,al
        jz      write_asciiz_string_x
        call    write_to_screen
        inc     si
        jmp     write_asciiz_string
write_asciiz_string_x:
        ret

;||||||||||||||||||||||||||||||||||||||||||||||||
;***************  #########  ****************
scroll_screen_up:
  ;routine to scroll the video screen up one line
        push    ax
        push    bx
        push    cx
        push    dx
        mov     ax,601H
        xor     cx,cx
        mov     dh,24
        mov     dl,79
        mov     bh,normal_attribute
        int     10H
        pop     dx
        pop     cx
        pop     bx
        pop     ax
        ret
```

```
;***********  @@@@@@@  ************
scroll_screen_down:
  ;routine to scroll the video screen down one line
   ;use BIOS call to scroll screen down
        push    ax
        push    bx
        push    cx
        push    dx
        mov     ax,701H
        xor     cx,cx
        mov     dh,24
        mov     dl,79
        mov     bh,normal_attribute
        int     10H
        pop     dx
        pop     cx
        pop     bx
        pop     ax
        ret

;&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
;***********  &&&&&  ***********
save_screen:
  ;routine to save video RAM for later recall
  ;this routine only has one video buffer
  ;and can only save one screen of data
  ;save video buffer with BIOS calls
        push    ax
        push    bx
        push    dx
        push    si
        push    di
        lea     di,screen_buffer
        mov     si,cursor
        mov     save_cursor,si
        xor     dx,dx
save_screen_loop:
        call    set_cursor_position
        mov     ah,8
        mov     bh,0
        int     10H
        cld
        stosw
        inc     dl
        cmp     dl,80
```

```
        jb      save_screen_loop
        mov     dl,0
        inc     dh
        cmp     dh,25
        jb      save_screen_loop
        mov     dx,si
        call    set_cursor_position
        pop     di
        pop     si
        pop     dx
        pop     bx
        pop     ax
        ret

;***********  ////// \\\\\  ************
restore_screen:
  ;restore video buffer from video save buffer
        push    ax
        push    bx
        push    cx
        push    dx
        push    si
        lea     si,screen_buffer
        xor     dx,dx
restore_screen_loop:
        call    set_cursor_position
        cld
        lodsw
        mov     bl,ah
        mov     ah,9
        mov     bh,0
        mov     cx,1
        int     10H
        inc     dl
        cmp     dl,80
        jb      restore_screen_loop
        mov     dl,0
        inc     dh
        cmp     dh,25
        jb      restore_screen_loop
        mov     dx,save_cursor
        call    set_cursor_position
        pop     si
        pop     dx
        pop     cx
```

```
        pop     bx
        pop     ax
        ret

video_routines  endp
;***************************
        end
```

Section 10

VIDEO GRAPHICS INTERFACE

INTRODUCTION This section explains some basics about the different standard PC video systems. It discusses pixels, palettes, and some video BIOS calls.

PICTURE ELEMENTS Pixel or Pel are standard references for a picture element. A pixel is the smallest programmable unit of a video display system picture. The number of bits used to define a pixel determines the number of possible colors you can select from when programming the pixel. If one bit is used for a pixel, you can select between two colors. If two bits are used for a pixel, you can select between four colors. If three bits are used for a pixel, you can select between eight colors. If four bits are used for a pixel, you can select between 16 colors, and so on. The coordinates of pixels are described by column and row with position column 0, row 0 being the leftmost column position of the topmost horizontal line. For a system of 640 columns by 200 rows, the last position on the bottom row is column 639, row 199.

VIDEO RAM FORMATS In some systems, the bits of a specific pixel are all in a single byte. In other systems, the bits of a specific pixel are spread across as many different bytes as there are bits for a pixel. When the pixel bits are spread across many bytes, the video memory is normally divided into what

are called memory planes. In some systems, each memory plane may be assigned to a specific color.

COLOR SELECTION AND PALETTES In some systems, a specific bit pattern for a pixel always displays as a specific color. In other systems, a specific bit pattern for a pixel is used to index a specific palette register. Each palette register may be programmed by software to select between a variety of colors to display. With palette registers, the color of an object on the display can be changed without redrawing the object image on the screen by changing the palette registers used by the object image.

In a VGA system, a palette register is called a digital-to-analog converter register (DAC). There are 256 DACs that are 18 bits wide each. The 18 bits are divided into three color groups with six bits of data for each color. This allows for 64 levels of each color. The three primary colors are red, green, and blue.

When a programmer is trying to decide what color to use in the display for a program, it is often convenient to let the user select all character and color attributes to help resolve any inconsistencies among different systems.

MONOCHROME DISPLAY ADAPTER (MDA) This video system has no high resolution graphics mode. The system displays 80 characters by 25 rows and requires two bytes of video RAM per character. One byte is used to select from 256 possible standard ASCII characters to display. The other byte selects the display attribute of the character.

COLOR GRAPHICS ADAPTER (CGA) This video system supports character display modes and graphics display modes. The character display modes are 80 characters by 25 rows and 40 characters by 25 rows. The graphics display modes are 640 columns by 200 rows with two colors and 320 columns by 200 rows with four colors. The IBM PCJR has a graphics system that is similar to CGA but is located at a different address and provides more colors in the high

resolution mode. Direct addressing of video RAM in older CGA systems can cause a hashing effect or a snow effect.

MULTICOLOR GRAPHICS ARRAY (MCGA) This video system has the same basic modes of a CGA system but provides 256 colors in a 320 columns by 200 rows mode and two colors in a 640 columns by 480 rows mode. This system also uses color palette registers.

ENHANCED GRAPHICS ADAPTER (EGA) This video system supports character display modes and graphics display modes. The character display modes are 80 characters by 25 rows, 40 characters by 25 rows, and 80 characters by 43 rows. This system provides between two to 16 colors in a 640 columns by 350 rows display. The more RAM on the EGA board, the more colors available to select from.

VIDEO GRAPHICS ARRAY (VGA) This video system supports character display modes and graphics display modes. The character display modes are 80 characters by 25 rows, 40 characters by 25 rows, 80 characters by 43 rows, and 80 characters by 50 rows. The graphics modes provided include a 320 columns by 200 rows with 256 colors and a 640 columns by 480 rows with two colors using color palette registers.

PROFESSIONAL GRAPHICS ADAPTER (PGA) This video system provides a graphics mode of 640 columns by 480 rows with 256 colors out of a set of 12 bit palette registers. This system also provides some high level graphics functions.

OTHER GRAPHICS SYSTEMS There is a variety of unique graphics for PCs. A very popular and standard system is the Hercules graphics adapter. This system provides 720 columns by 348 rows in a monochrome mode. Unique and enhanced graphics systems have been made available by many companies that release PC clones.

THE INT 10H VIDEO FUNCTIONS The system software interrupt 10H is used for video BIOS functions. This section discusses the get CRT mode, set CRT mode, write pixel, and read pixel.

To set the CRT display mode, have AH set to 0 and AL set with CRT display mode code, then execute an interrupt 10H.

The following table has the code values in hex for the standard CRT graphics modes and the associated graphics resolution. These codes are used for get and set display mode functions.

04H	320 by 200, 4 color
05H	320 by 200, 4 color monochrome
06H	640 by 200, 2 color
0DH	320 by 200, 16 color
0EH	640 by 200, 16 color
0FH	640 by 350, 4 color
10H	640 by 350, 16 color
11H	640 by 480, 2 color
12H	640 by 480, 16 color
13H	320 by 200, 256 color

To check the current video display mode from a program, the video BIOS function call 15 to get the CRT mode can be used. To invoke the function, have register AH set to 15 and execute interrupt 10H. On exit from the interrupt, register AL has the current CRT mode, register AH has the number of columns on the screen, and register BH has the current active page number. This call can be used after a set CRT mode call to verify that the system is using the correct mode.

To write a new pixel value to the display, on entry have register AH set to 0CH, register DX set with the row number, register CX set with the column number, register AL set with the color value data, and register BH set to display page number (this value is normally set to 0 for graphics). Execute an interrupt 10H to invoke the function call.

To read the current value of a pixel, on entry have register AH set to 0DH, register DX set to the row number, register CX set to the column number, and register BH set to display page number (normally a value of 0). On return from the interrupt 10H function, register AL has the color value of the pixel.

The following code example shows how to draw a horizontal line in graphics mode.

```
.CODE
;draw line code
draw_line proc  near
        mov     AL,line_color
        mov     DX,line_row_start
        mov     CX,line_col_start
        mov     BX,0            ;display page
        mov     SI,line_length
        mov     AH,12
draw_line_loop:
        push    AX
        push    BX
        push    CX
        push    DX
        push    SI
        int     10H             ;video BIOS call
        pop     SI
        pop     DX
        pop     CX
        pop     BX
        pop     AX
        inc     CX              ;point to next line pixel position
        dec     SI              ;adjust line length counter
        jnz     draw_line_loop
        ret
draw_line endp
```

Section 11

COMPUTER MATH

INTRODUCTION This section explains some of the basics of doing math in Assembly language. There are several hardware and software tricks that can be used to perform math calculations as quickly as possible. This section discusses bit level multiplication with shifts and adds.

ADDING DATA There are two basic integer add instructions: the standard ADD and the ADC (add with carry). Normally, just the standard add is used for binary integers, but with some code, the add with carry is necessary. Both the ADD and the ADC instructions will alter the contents of the carry flag to indicate an addition overflow out of the high bit position, but the ADC will check the contents of the carry flag at the instruction start and add an extra 1 to the two operands being added if carry condition is true.

Example of a 32 bit binary integer addition:

```
;add lower half of 32 bit data to AX
  add     ax,datalow
;add upper half of 32 bit data to DX
  adc     dx,datahigh
;branch if carry overflow
  jc      carryover
```

SUBTRACTING DATA There are two basic subtract instructions: the standard SUB and the SBB (subtract with borrow). Normally, for small numbers, the SUB is used for standard binary integer subtraction, but in routines that want to handle subtractions of larger numbers, the subtract with borrow is used. The borrow condition is maintained in the carry flag.

Example of a 32 bit subtraction:

```
;subtract lower half of 32 bit data from AX
   sub     ax,datalow
;subtract upper half of 32 bit data from DX with borrow
   sbb     dx,datahigh
```

MULTIPLYING DATA There are two basic multiply instructions: the integer unsigned multiply (MUL) and the integer signed multiply (IMUL). The multiply can be either 8 bit or 16 bit format. For the 8 bit multiply, AL has to hold one data variable. The other data variable can come from a register or memory. The 16 bit result will be placed into AX. For the 16 bit multiply, AX has to hold one data variable. The other can come from a register or memory. The 32 bit result will be placed into DX:AX with DX holding the most significant data and AX holding the least significant data.

Examples:

```
;multiply BX * AX = DX:AX
    MUL     BX
;multiply data to accumulator
    IMUL    data_var
```

Note that if data_var is a byte, the compiler will generate code for an 8 bit by 8 bit multiply instruction; if data is a word, then it will generate a 16 bit by 16 bit multiply instruction.

DIVIDING DATA There are two basic divide instructions: the standard integer unsigned divide (DIV) and the integer signed divide (IDIV). You can divide a 16 bit number by an 8 bit number or you can divide a 32 bit number by a 16 bit number. For the small divide, the AX register has to hold the 16 bit number. The 8 bit number that is used for the small divide can come from either a register or memory. The result

will be in AL with the modulo from the divide in AH. For the larger divide, DX:AX holds the 32 bit data with AX holding the least significant bits. The 16 bit data used for the divide can come from a register or memory. The main result data will be put into AX and DX will hold the modulo data resulting from the divide.

Example:

```
;divide DX:AX by BX = AX modulo DX
       DIV     bx
;divide using signed integer AX by databyte = AL modulo AH
       IDIV    databyte
```

MULTIPLYING BY LEFT BIT SHIFTING Shifting a number to the left by one bit position has the effect of multiplying the number by two. Shifting a number to the left by two bit positions has the effect of multiplying the number by four. Shifting a number to the left by three bit positions has the effect of multiplying the number by eight. Each shift doubles the current value of a binary integer variable. In base ten math, shifting a number to the left and putting a zero in the rightmost digit position multiplies the number by ten.

A more elaborate multiplication can be seen with this example. Shift a number left one bit to multiply by two and save the number. Now shift the number left two more times to multiply by eight, then add the saved value of the two multiply and you have the effect of multiplying by ten. Now take this result and shift to the left one more time to double the data value and you have multiplied the starting number by 20.

Example:

```
mov     ax,01   ;load 1 into ax
shl     ax,1            ;ax now 10B or 2
mov     dx,ax   ;save value
shl     ax,1            ;ax now 100B or 4
shl     ax,1            ;ax now 1000B or 8
add     ax,dx   ;ax now 1010B or 10
shl     ax,1            ;ax now 10100B or 20
```

INCREMENTING AND DECREMENTING The INC instruction is a quick way to add one to memory or a register. This is used in many routines that count. The DEC instruction is a quick way to subtract one from memory or a register.

```
;add to accumulator
      inc     ax
;subtract 1 from memory location data
      dec     data
```

NOTES ABOUT USING THE 80X87 MATH COPROCESSOR The 80X87 coprocessor is a very powerful math processing unit. It is an optional math processor that is available for most PC systems. The 80X87 has eight internal data registers for processing math functions that are separate from the standard 80X86 registers. Each of the eight 80X87 data registers is 80 bits wide. The 80X87 has several input and output formats: two byte word integer, four byte short integer, eight byte long integer, ten byte packed decimal, four byte short real, eight byte long real, and a ten byte temporary real. All data inside of the 80X87 is handled in the same 80 bit real number format for math functions. The 80X87 system uses a stack pointer system to index the internal data registers. The 80X87 data registers are addressed as relative to their current position in the stack. The stack works in a circular motion from indexing data register 0 to data register 7 and back to data register 0. The data at the top of the stack is referenced as ST(0). If you pop data off the stack, then ST(1) becomes ST(0). If you push data into the stack, then ST(0) becomes ST(1). The 80X87 is designed to run in parallel with the 80X86 CPU. An interesting note about 80X87 math is the division by zero which results in an answer of infinity except when zero is divided by zero which is indefinite.

Section 12

C LANGUAGE INTERFACE

INTRODUCTION This section discusses how to make Assembly language code work with a Borland C language program. It explains some of the internal workings of the C language and some fundamental ideas about a compiled C language program's data area. It is necessary to understand these concepts in order to write Assembly language routines that can interface efficiently with a C language program. The Turbo Assembler development system provides many functions for writing Assembly language code to interface to C language code and other high level languages. The Borland C++ development system allows for Assembly language code to be inserted directly into a C language program or for separate Assembly language code to be linked into a C language program.

C LANGUAGE DATA AREAS The data for C code functions and routines are based on heap and stack structures. To explain this in a simple way, assume that a program is assigned one contiguous block of memory to use upon execution. The code of the program is loaded into the lowest address space of the block of memory. The heap data is loaded directly above the code in lower memory and expands up as necessary for more heap data space. The stack data starts at the top of the memory block and expands down towards the heap data as more stack data space is needed. Let us hope that the heap

data and the stack data shall never meet because this may crash a program.

A general method to access data passed to a routine from a standard C function is to use the BP register. The following code segment shows how to define BP at the start of an Assembly language code function and addresses the parameters passed from a C language program.

```
        .MODEL  small
        .CODE
        PUBLIC  _Test_Asm
_Test_Asm PROC
        push    bp              ;save old BP value
        mov     bp,sp   ;set BP to index local data
        mov     ax,[bp+4]       ;load AX with parameter 1
        add     ax,[bp+6]       ;add parameter 2 to AX
;exit procedure
        pop     bp              ;restore old BP value
        ret                     ;exit to C program with AX = P1 + P2
_Test_Asm ENDP
        END
```

The following is an example of a C language code routine to execute the _Test_Asm Assembly language function.

```
extern "C" { int Test_Asm( int, int); }
main() {
int sum;
Sum = Test_Asm(1, 2)
}
```

MEMORY MODELS When programs become large and either the code, stack, or data segments grow to exceed 64KB, then certain problems develop. Because of speed along with code size problems and 64KB problems, the Turbo Assembler system provides several basic memory models for code development. They are: tiny, small, medium, compact, large, huge, tchuge, tpascal, and flat. The basic differences between the different memory models have to do with assumptions about the data being addressed with near or far pointers and code routines being called as near routines or as far routines. Many of the problems are solved with special compiler functions provided in a Borland C++ development system.

The following is a list of C data types and the associated Asembler data types.

unsigned char	byte
char	byte
enum	word
unsigned short	word
short	word
unsigned int	word
int	word
unsigned long	dword
long	dword
float	dword
double	qword
long double	tbyte
near*	word
far*	dword

The following is a list of C language function return data types and the Assembly language registers that the data types use.

unsigned char	AX
char	AX
enum	AX
unsigned short	AX
short	AX
unsigned int	AX
int	AX
unsigned long	DX:AX
long	DX:AX
float	8087 ST(0)
double	8087 ST(0)
long double	8087 ST(0)
near*	AX
far*	DX:AX

INLINE ASSEMBLY CODE FOR BORLAND C Assembly language code can be written directly into a C language program with the asm function. When using this function, your Assembly language code is limited to a subset of the functions provided by the Turbo Assembler system. The following exam-

ple shows the use of the asm function in a C language program segment.

```
int Var1, Var2;
Test_Asm()
{ asm { mov ax,Var1
        add ax,Var2
      }
}
```

COMPILING C AND ASSEMBLY TOGETHER Borland C++ provides a convenient way to compile C++ language programs with Turbo Assembly language routines. The programmer can compile both programs together with a single compile statement as in the following example.

```
BCC source1.cpp source2.asm
```

This DOS command statement will compile the C++ source code (source1.cpp) into an object module, then invoke the Turbo Assembler to compile the Assembly code (source2.asm) into an object module, then invoke TLINK to link the modules into an executable program.

To see how a C language program looks in Assembly language, you can compile a C program with the -S option to generate an Assembly output program listing.

UNDERSCORES AND NAMING CONVENTIONS The Assembly language code needs to add an underscore character to the start of function names to be called from a C language program. For example, if the C language program makes a call to an Assembly language function called asm_fun in the C language program, then the Assembly language program should use the name _asm_fun for the Assembly language code. The name of the Assembly language function that the C language program calls must be case sensitive. The compiler option /ml can be used to make an Assembly language program case sensitive. The Assembly language function name needs to be declared as PUBLIC.

Appendix A

GLOSSARY OF TERMS

ASCII	An international standard for character codes to promote the portability of character string data between different computer systems.
Assembler	A compiler that takes an Assembly language source code file and outputs a machine language object code file.
BIOS	The basic input output system software provided by the computer to handle hardware interface.
Bug	A condition when the computer system does not operate according to specifications.
Conditional Assembly	This is a method of using a compiler directive to either compile or ignore a section of code. A variable used to decide if code is compiled or not can be set just before or at compile time. This feature is used by programmers for many different reasons. A common use of conditional assembly is to generate special code that is used only for debugging versions of a program.
Contiguous	A block of computer memory where a program has control of all memory variables from the start of the memory area to the end

	of the memory area. This means that no other program can use any segment of the memory inside of a contiguous memory area of a program.
CPU	The central processing unit of a computer system. This is the hardware unit that executes a computer program.
Debugger	A tool that may be hardware or software that is used to aid in finding computer bugs.
DOS	The disk operating system of a computer system.
Macro	A macro is in some ways like a subroutine except a macro is expanded in the code each time the macro is referenced, which removes the need for call and return instructions. A macro executes faster than a subroutine but requires more memory space if referenced more than once. If you have a subroutine that is only referenced once, you may want to convert it into a macro to speed up the code by removing the call and return instructions.
Mnemonic	The Assembly language instruction names are often referenced as the instruction mnemonics. Example: MOV is the mnemonic for the move instruction.
Operand	The data that is used for a computer instruction.
Peripheral	A device that is attached to a computer system such as a printer, modem, etc.
Register	A special memory location inside a CPU. There are many registers inside a standard CPU. Many of these registers serve specific functions for instruction executions and CPU operations.

Appendix B

DEFINING DATA

The structure and organization of your data can be very important to the effectiveness of your program code. This is true for any computer language.

One of the most common ways to define data is with a DB statement. This statement is used to define data in byte format. It may be used to define one byte or a string of bytes. You can define numeric data or ASCII data. It is very flexible.

```
   ;define one byte with value zero
        db      0
   ;define one byte with value unknown
        db      ?
   ;define an ASCII text string of bytes
        db      "define many byte string of ASCII text"
   ;mix up data types in statement
        db      0,?,"mix up data statement"
   ;define binary bit pattern for 18H
        db      00011000B
   ;define an ASCIIZ string with label
text_to_print   db      'This is ASCIIZ string',0
   ;define data block of zeros that is 300 bytes big
arrayX  db      300 dup(0)
```

The DW statement is used in many data definitions. It is used to define 16 bit words of data. This data statement is also just as flexible as the DB statement. These words can be used as indirect jump vectors. When data structures are addressed as words, the CPU speed is dependent on the alignment of the

word being at an even or odd address. A word with an even address will be processed faster than a word with an odd address.

```
        even            ;force even word addresses
data    dw      1       ;define word with value one
   ;define array of 80 words with zero
array80 dw      80 dup(0)
```

If a word is addressed as two bytes, this will appear with the least significant byte at the first memory address and the most significant byte at the next memory address.

The DD statement is for defining 32 bit double words. These double words can be used for indirect far jump vectors.

```
        dd      offset_data:dataseg
```

There are other data definition type statements that are sometimes used. The following are examples of some standard data types.

df	6 byte farword
dq	quad word, 8 bytes
dt	ten byte, 8087 format

Part of writing a complete Assembly language routine for the INTEL 80X86 processor requires the program to have an assume statement. This is used by the compiler to detect segment addressing errors. It is the programmer's responsibility to make sure that the segment registers are indexing the correct data area at any given time. Using compiler directives provided by the Turbo Assembler system, the programmer can simplify and avoid the use of many assume statements.

The following example of bad code is used to illustrate how some addressing problems can occur.

```
DataSeg1        SEGMENT para    public  'data'
var_1   dw      0
DataSeg1        ends
DataSeg2        SEGMENT para    public  'extra'
var_2   dw      7
DataSeg2        ends
        assume  CS:CodeSeg
CodeSeg SEGMENT para    public  'code'
start   PROC    near
```

```
    ;this gets the address of DataSeg1
       mov     ax,DataSeg1
    ;this loads DS to index DataSeg1
       mov     ds,ax
       assume  DS:DataSeg1
;this next mov statement will generate a compile error
  ; because var_2 is not in DataSeg1
       mov     ax,var_2
```

Data, whether it is in a register or in memory, can be viewed by the program code in one of two ways: as a working data value or as a pointer to a data value. There are many types of pointers and the 80X86 allows for some complex pointing support. In the 80X86, there are segment registers which are used as base pointers to index the start of a memory segment. Almost all instructions that address memory will have a segment register implied or declared for calculating the real memory address to use. Many of the CPU registers can be used to point to data as well as hold data values. Data pointers may be direct or indirect, that is, a pointer may directly point to the location of a data value or it may point to the location of another pointer.

Appendix C

MOVING DATA

This appendix covers the most commonly used instructions for moving data. The instructions discussed include the MOV, PUSH, POP, IN, and OUT.

The MOV statement is generally used to move data in and out of registers. It can also be used to move data from memory to memory and to load memory or registers with immediate data. When moving data, always define the destination first, then the source.

The following are some examples of MOV statements.

```
;load register AX with value 55
        mov     AX,55
;this loads AX with contents of BX
        mov     ax,bx
;this loads AX with contents of memory word indexed by DS:BX
        mov     ax,[bx]
;this loads AL with contents of memory byte indexed
  ;by SS:BP+BX
        mov     al,[bp+bx]
;define a word of memory for use as a variable v_1
v_1     dw      0
  ;load AX with data from v_1
        mov     AX,v_1
  ;load AX with data from v_1
        mov     AX,[v_1]
  ;move AX to memory location v_1
        mov     v_1,AX
```

```
;move 55 to memory location v_1
      mov     v_1,55
```

There are special instructions to move data from one memory location to another memory location. These are the string move instructions: MOVSB is used for byte movements and MOVSW is used for word movements. These instructions can be used with or without the REP (repeat) instruction. If used with the REP instruction, the direction flag bit must be set along with the count in register CX before executing the REP. If CX is set to zero at the start of a repeat, then it will loop 65536 times. The string move instructions must use DS:SI as the source memory location and ES:DI as the destination memory location. After each byte or word move with a repeat instruction, the SI and DI registers are altered to index the next location.

```
      cld     ;clear direction bit for backward
      std     ;set direction bit for forward
;setup index pointers and segments
      mov     es,destination_segment
      lea     si,source_string
      lea     di,destination_string
;move string byte from SI to DI
      movsb
;move string word from SI to DI
      movsw   ;move string word from SI to DI, dec CX if
;set up counter for repeat
      mov     cx,10
;move 10 words from SI to DI
      rep     movsb
```

The IN and OUT instructions are used by the 80X86 to address peripheral devices such as interrupt controllers, video controllers, communications ports, etc. When addressing an I/O port, the DX register is commonly used as the port address index. If an eight bit port number (0 - 255) is used, it can be addressed directly. The basic 8086 can only IN and OUT eight bits of data at a time and the AL register is always either the source or destination. Other processors in the 80X86 line may use 16 bits at a time. The following is an example of I/O port addressing.

```
      IN      al,20H
      OUT     20H,al
```

```
;For all 16 bit port addresses,
  ; the DX register must be used as the index.
        mov     dx 3BDH
        in      al,dx
        mov     al,40H
        out     dx,al
```

PUSH and POP are used to save the contents of registers into a temporary stacking area for recall at a later time. This works like a LIFO (Last In First Out) structure which allows for systematic nesting of data variables. This can be important to routines that must not destroy the original contents of the registers during their execution and also for routines that need to be reentrant. Hardware interrupting routines are an example of this.

Many high level languages use stack frame structures to store data. To address data in these stack frame structures, the BP register can be used because it is the natural stack data index. The following code displays how this may be done.

```
Start   proc    near
        push    bp
        mov     bp,sp
 ;to access the last word pushed on stack before a near call
   ; was made to this procedure use offset of 4 to BP
        mov     ax,bp+4

Start   proc    far
        push    bp
        mov     bp,sp
 ;to access the last word pushed on stack before a far call
   ; was made to this procedure use offset of 6 to BP
        mov     ax,bp+6
Start   proc    near
        push    bp
        mov     bp,sp
        sub     sp,8
```

The subtract instruction opens up four words of data space for the procedure to use as temporary variable space. The data can be addressed by using the offsets of BP-2 to BP-8. On exit from these routines, you must remember to reset the stack pointer back before you can execute the return instruction correctly.

The exchange instruction is used as a quick way to swap data between registers or registers and memory.

```
XCHG    ax,bx
XCHG    bx,data
```

The translate instruction (XLAT) is used to translate eight bit data codes. You can translate EBCDIC to ASCII with the correct translation table set up. The BX register is used to index the base address of a 256 byte data block. The AL register is added to BX to get a byte address of the data to load into the AL register.

The load effective address instruction is used to get the address of a data location or an execution routine. This is very useful when you want to get the address of some data variable to pass to another routine for processing.

```
variable_string db      'This is a variable data string',0
        LEA si,variable_string
```

Appendix D

JUMPING INSTRUCTIONS

Jumping instructions are a unique set. They are the ones that reset the IP register or the CS:IP register pair. By doing so, they cause an alteration of the standard next instruction program sequence. The CS:IP register pair is used by the CPU to keep track of where the next instruction to execute is located.

STANDARD JUMPS In the standard 80X86 instruction set, there are three different size offset values that can be used for jumping. They are the 8 bit short jumps, the 16 bit near jumps, and the 32 bit far jumps. The 32 bit far jumps use two 16 bit words combined to make up a 20 bit segment:offset address. The following are some common examples of jump instruction statements.

```
;goto near procedure next_task
        jmp     next_task
;goto far procedure next_task
        jmpf    next_task
;the programmer does not need to use the jmpf mnemonic
   ; because the compiler will decide and use when needed
load BX with location of code routine
        mov     BX,offset next_task
        jmp     [BX]            ;goto where BX is pointing
;define data word with address of code routine
next    dw      offset next_task
        jmp     [next]          ;goto next code routine
```

CONDITIONAL JUMPS A standard conditional jump instruction checks the contents of the Flag register bits relating to the jump condition. If the associated bits are set correctly for the jump condition, then the IP register is altered and program execution continues at the new address indicated by the instruction data. All conditional jump instructions use an eight bit jump offset. To the programmer, this means that you can only jump over a few instructions. You can usually jump over 30 to 40 instructions; beyond that, you take a chance with compiling errors.

The following is a list of jump mnemonics with conditions and explanatory notes:

JB jump below

This is normally used after a compare to see if the first variable was logically below the second value using unsigned numbers. Same as JC instruction.

JBE jump below or equal

This is normally used after a compare to see if the first variable was logically below or equal to the second value using unsigned numbers.

JA jump above

This is normally used after a compare to see if the first variable was logically above the second value using unsigned numbers.

JAE jump above equal

This is normally used after a compare to see if the first variable was logically above or equal to the second value using unsigned numbers.

JC jump carry

This is normally used to test for a set carry bit in the Flag register after an addition overflow or subtraction borrow. Same as JB instruction.

JNC jump no carry

This is normally used to test for a no carry bit in the Flag register after an addition overflow or subtraction borrow.

JL jump less

This is normally used after a compare to see if the first variable was less than the second value using signed numbers.

JLE jump less or equal

This is normally used after a compare to see if the first variable was less than or equal to the second value using signed numbers.

JG jump greater

This is normally used after a compare to see if the first variable was greater than the second value using signed numbers.

JGE jump greater or equal

This is normally used after a compare to see if the first variable was greater than or equal to the second value using signed numbers.

JS jump sign

This jump is used for testing for a set sign bit in the Flag register. The sign bit is set during some logical instructions to the same value as the most significant bit of the data.

JNS jump no sign

This jump is used for testing for a no sign bit in the Flag register. The sign bit is set during some logical instructions to be a copy of the most significant bit of the data.

JO jump overflow

Branch is taken if the overflow bit in the Flag register is set.

JNO jump no overflow

Branch is taken if the overflow bit in the Flag register is not set.

JP jump parity

Jump if the parity bit in the Flag register is set. Same as JPE instruction.

JNP jump no parity

Jump if the parity bit in the Flag register is not set. Same as JPO instruction.

JCXZ jump CX zero

This is a special condition jump for quick testing of the CX register and branching if zero. This does not test or change any bits in the Flag register.

LOOP decrement CX and jump if CX not zero

This is a special condition jump normally used in repeat structures; it decrements the CX register and jumps if CX is not zero. Note that if you start with the CX register equal to zero, it loops 65536 times. To prevent this condition, the JCXZ instruction can be used before looping logic begins.

Real-time programmers note that the logical branch instructions execute several times faster if the jump is not taken. If possible, write your code such that the jumps are not taken under most conditions.

SUBROUTINE JUMPS When you call a subroutine, the CPU saves the current contents of the CS:IP register pair or the IP register in the stack area so that when the subroutine is finished, it can load the return address from the stack area into the IP register or CS:IP register pair for continuation at the next instruction after the call statement.

```
;call procedure, compiler will decide if near or far
      call    subroutine
;call using address in memory location
      call    [indirect]
;call using address in register BX
      call    [BX]
```

A subroutine call is terminated by a matching return instruction. The return instruction is a special case load instruction that loads the IP register or the CS:IP register pair with data indexed by the stack pointer.

RET return

This is a standard near return that only pops the IP register from the stack. This is to be used by subroutines that are defined as near procedures.

RETF return far

This is used when a subroutine is defined as a far procedure to return. This pops the CS:IP register pair off the stack. The reason that all calls are not far calls is to save memory and speed. Normally, the programmer will just use the RET mnemonic and let the compiler decide if the return is near or far.

INTERRUPT JUMPS These are normally used for calling system functions.

INT interrupt

This is a software interrupt instruction that jumps to a location determined by the data and a vector table lookup. The interrupt vector table occupies the first 1,024 bytes of CPU addressing range. There are 256 different interrupt vectors. Each vector is made up of four bytes of memory locations for interrupt data to define CS:IP to the start of the interrupt routine.

IRET interrupt return

This is like a normal RETF except the Flag register is popped as well as the CS:IP register pair. Note that you can make a far call look like an interrupt call by pushing the Flags before making the far call.

Because the conditional jumps are only eight bit offsets, problems will occur if you attempt to jump over more than a few instructions. One standard way to overcome this problem is to translate the jump as follows:

```
original
        jc      carry_overflow
        nop
carry_overflow:
```

translates to

```
        jnc     no_overflow
        jmp     carry_overflow
no_overflow:
carry_overflow:
```

TABLE LOOKUP JUMPING If you have a data variable and you want to execute one of several different functional routines depending on the data, then the table lookup jump is a good method for branching control. There are many ways that table lookup jumping can be performed, depending on the nature of the data that is used to select a particular subroutine from a set of possible subroutines.

The following is an example to select a jump given a positive binary integer between 0 and 32000:

```
function_table  dw      function_0_routine
                        dw      function_1_routine
                        dw      function_2_routine
  ;load BX with number
        mov     bx,function_code
  ;point DI to index start of jump table
        lea     di,function_table
  ;adjust BX to word offset into jump table
        shl     bx,1
  ;perform call to routine indexed in table
        call    [di+bx]
```

The following is an example of keyboard table lookup jumping.

```
keyboard_key            dw      0031
function_key_table      dw      0031
                        dw      function_1_routine
table_end:
;
;load AX with function code
        mov     ax,keyboard_key
;index jump table with SI
        lea     si,function_key_table
function_loop:
        cmp     si,table_end            ;?? end of table ??
        jae     no_find_function        ;branch if end
     ;check for match of function codes
        cmp     ax,[si]
```

```
        je      function_find           ;branch if find code
        add     si,4                    ;else point to next entry
        jmp     function_loop
function_find:
        jmp     [si+2]
no_find_function:
```

Appendix E

LOGICAL INSTRUCTIONS

This appendix discusses the basic logical instructions and details the operations they perform on data.

COMPARING DATA The compare instruction works the same as the subtract instruction except that the resulting data variables are thrown away. What is important about the compare instruction is that the Flag register is reset by the operation. This allows for conditional branching following the compare instruction. There are 8 bit and 16 bit compares.

Example:

```
;compare AX to data (AX-data)
     cmp     ax,data
;compare data to AX (data-AX)
     cmp     data,ax
;compare AX to 11 (AX-11)
     cmp     ax,11
;compare data to 11 (data-11)
     cmp     data,11
```

Note that you can compare a sign extended immediate 8 bit data value to a 16 bit data value.

Example:

```
     cmp     ax,1
```

STRING INSTRUCTIONS String instructions can be used with the repeat command. When using the repeat command (REP), you must always preset the direction flag and register CX with the count. There are five standard string commands that can work in either byte mode or word mode. The five commands are MOVS, CMPS, SCAS, LODS, and STOS. The move string (MOVS) is explained in the appendix on moving data. The string compare (CMPS) is used to compare two different strings of data in memory to each other. To use the string compare instructions, you must set DS:SI to index source memory location and ES:DI to index destination memory location. After each byte or word compare with a string instruction, the SI and DI registers are altered to index the next location. If a repeat command is used with a string instruction, the CX register is decremented by one and then tested for zero to determine if the loop should continue. There is an instruction to compare the contents of memory to the accumulator (SCAS). The LODS is used to load the accumulator with a string of data values from memory. The STOS is used to store the contents of the accumulator to a string of memory.

Examples:

```
  ; define string locations for comparing
string1 db      'test string ',0
string2 db      'test string ',0
string_size     equ     12
   ;get ready for string compare
        cld     ;go forward
        mov     cx,string_size
        lea     si,string1
        lea     di,string2
   ;test to see if two strings are equal
        repe    cmpsb
   ;branch if strings donot match
        jne     strings_donot_match
strings_donot_match:

   ;find end of ASCIIZ string
        cld     ;search forward
        mov     cx,200  ;set to max string size
     ;set accumulator to zero for end of string character
        mov     al,0
```

```
        lea     di,string2
      ;repeat search until zero found or CX = 0
        repne   scasb
      ;branch if no end of string found
        jne     no_end_of_string
no_end_of_string:

   ;set block of memory to zero
        cld     ;go forward
        mov     cx,6    ;size of block in words
        lea     di,string1
        mov     ax,0
      ;write zeros to block
        rep     stosw
```

BIT MANIPULATING INSTRUCTIONS These instructions are used to deal with data one bit at a time. They are used in many graphics routines for controlling the bit patterns of video arrays. The basic instructions are the logical Boolean operations.

```
   ;AND accumulator with 0000 0000 0000 1111 bit pattern
        AND     ax,0FH
   ;OR memory data with bit pattern in accumulator
        OR      data,ax
   ;eXclusive OR accumulator with bit pattern in register DX
        XOR     ax,dx
   ;reverse bit pattern in accumulator
        NOT     ax
```

TESTING DATA The TEST instruction is the same as the AND instruction but the result only affects the Flag register without affecting any of the operand data values.

Example:

```
        test    al,80H          ;Test high bit
```

BIT SHIFTING INSTRUCTIONS The basic logical and arithmetic shift instructions have the same effect as multiplying or dividing a number. If you take a binary number and shift all the bits over one position right, this is the equivalent of multiplying by 0.5 or dividing by 2, which is the same as cutting the value of a binary number in half.

SHR logical shift right, move zero into high bit position and move low bit into carry flag.

SHL logical shift left, move zero into low bit position and move high bit into carry flag.

SAR arithmetic shift right, keep same value in the high bit position and move low bit into carry flag.

SAL arithmetic shift left, move zero into low bit position and move high bit into carry flag.

ROL rotate left, move high bit position to low bit position.

ROR rotate right, move low bit position to high bit position.

RCL rotate left through carry flag, move high bit data to carry flag and move carry flag data to low bit position.

RCR rotate right through carry flag, move low bit data to carry flag and move carry flag data to high bit position.

NO OPERATION When debugging code at the Assembly language level, you will probably encounter NOP instructions inserted into the executing code. The NOP instruction is used for many reasons. Most of these reasons have to do with compiler problems. In complex instruction sets, there are variable length instructions. These instructions create problems for compilers that must allocate space for the instruction without knowing the exact size of the final instruction. For example, there are near CALLs and far CALLs. When the compiler runs into a subroutine CALL, it may not know if it is a near CALL or a far CALL. The near CALL instruction will only need three bytes and the far CALL instruction will need five bytes. The compiler must assume worst case and assign space for the far CALL even though it may be a near CALL. If it turns out to be a near CALL, then there is the problem of what to do with the space not used. A solution is to insert NOP instructions into the unused spaces. This is one reason why all computers with complex instruction sets have a no operation instruction.

Appendix F

INTERFACING TO BIOS

The most commonly used keyboard, video, and printer BIOS calls are detailed in this appendix. When interfacing to the BIOS routines, the programmer uses interrupt calls. The specific interrupt number to be used depends on the function. The keyboard BIOS calls use INT 16H; the video BIOS calls use INT 10H; the printer BIOS calls use INT 17H. To use these BIOS calls, you set AH with a function code for a specific request and then interrupt into the routine with the INT instruction.

BASIC KEYBOARD CALLS Use interrupt INT 16H for all keyboard BIOS functions.

Read Keyboard

On entry, AH=0.

On exit, data is in AX. The value in AX can be subdivided into AL data and AH data. If the data in AL is from 1 to 255, then it is a standard ASCII key code with the value in AH being a specific keyboard scan code. If you are testing for an ASCII key code, then the only data you need to look at is in register AL. If the value in AL is zero, then AH holds a special function key code value. This means that a function or special key was pressed.

Scan Keyboard

On entry, AH=1.

On exit if the Z condition is set, then there is no current keyboard data, else AX has keyboard scan data. If the NZ condition is set, to remove the data from the keyboard buffer, you must call the read keyboard function.

Get Current Shift Status

On entry, AH=2.

On exit, special keyboard data is in AL,

bit 0 - Right shift,

bit 1 - Left shift,

bit 2 - Ctrl,

bit 3 - Alt,

bit 4 - Scroll active,

bit 5 - Num lock active,

bit 6 - Caps lock active,

bit 7 - Insert state active.

BASIC VIDEO CALLS Use interrupt INT 1OH for all video BIOS functions.

Set CRT mode

On entry, AH=0,

if AL=0 then set 40x25bw text mode,

if AL=1 then set 40x25co text mode,

if AL=2 then set 80x25bw text mode,

if AL=3 then set 80x25co text mode,

if AL=4 then set 320x200co graphics mode,

if AL=5 then set 320x200bw graphics mode,

if AL=6 then set 640x200bw graphics mode,

if AL=7 then set 80x25bw text mode.

There are additional modes discussed in greater detail in the video graphics interface section.

Set Cursor Type

On entry, AH=1, and CX has cursor type data where CH has start line for the cursor in bits 0-4 and CL has end line for the cursor in bits 0-4. The specific height of a cursor cell is dependent on the video mode. The other bits (5-7) should be set to zero.

Set Cursor Position

On entry, AH=2, BH=video page number, DH=row value, DL=column value.

Get Cursor Position

On entry, AH=3, BH=video page number.

On exit, DH=row position, DL=column position, CX=cursor type data.

Read Light Pen Position

On entry, AH=4.

On exit, AH=0 if no input data or if AH=1 then DH=character row, DL=character column, CH=raster line, BX=pixel column.

Select Active Page

On entry, AH=5, if AL<128 then AL=new page,

if AL=80H read crt/cpu registers,

if AL=81H set cpu register with BL,

if AL=82H set crt register with BH,

if AL=83H set cpu/crt with BH, BL.

On exit, BH=crt register, BL=cpu register.

Scroll Up

On entry, AH=6, AL=line scroll count, if AL=0 make blank page, BH has attribute data for new blank lines, CX and DX are used to hold data for scroll window frame, CH=upper left row, CL=upper left column, DH=lower right row, DL=lower right column.

Scroll Down

On entry, AH=7, AL=line scroll count, if AL=0 make blank page, BH has attribute data for new blank lines, CX and DX are used to hold data for scroll window frame, CH=upper left row, CL=upper left column, DH=lower right row, DL=lower right column.

Read Character & Attribute

On entry, AH=8, BH=video page number.

On exit, AL=character data, AH=attribute data.

Write Character & Attribute

On entry, AH=9, AL=character data, BH=video page number, BL=attribute, CX=count of write.

Write Character

On entry, AH=10, AL=character data, BH=video page number, CX=count of write.

Set Color Palette

On entry, AH=11,

if BH=0 then set background color to BL,

if BH=1 then set default palette to number in BL.

If bw mode BL=0 for white, BL=1 for black.

If four color CGA mode, BL=0 for black, green, red, yellow or BL=1 for black, cyan, magenta, white.

Write Dot

On entry, AH=12, AL=dot color data, DX=row data, CX=column data.

Read Dot

On entry, AH=13, DX=row data, CX=column data.

On exit, AL=dot color data.

Write TTY

On entry, AH=14, AL=character data, BL=foreground color data. When using this DOS call, ASCII control characters such as the TAB, FormFeed, LineFeed, etc. will not print but will cause the ASCII control function to be performed.

Get CRT Mode

On entry, AH=15.

On exit, AL=video mode, AH=number of columns, BH=active page number.

Set Palette Registers

On entry, AH=16, if AL=0, BL=number of palette, BH=color data, if AL=1, BH=boarder color, if AL=2 set palette color values.

BASIC PRINTER CALLS Use interrupt INT 17H for all printer BIOS functions. On return from a print character call, you need to check the printer return status to make sure that the character did print.

Print character

On entry, AH=0, AL=character data, DX=printer port number.

On exit, AH has status,

bit 0 - Timeout status,

bit 1 - not used,

bit 2 - not used,

bit 3 - I/O error occurred,

bit 4 - Selected status,

bit 5 - Out of Paper Error,

bit 6 - Acknowledge status,

bit 7 - Not busy status.

Reset printer port

On entry, AH=1, DX=printer port number.

On exit, AH has status,

bit 0 - Timeout status,

bit 1 - not used,

bit 2 - not used,

bit 3 - I/O error occurred,

bit 4 - Selected status,

bit 5 - Out of Paper Error,

bit 6 - Acknowledge status,
bit 7 - Not busy status.

Get Current Printer Status

On entry, AH=2, DX=printer port number.
On exit, AH has status,
bit 0 - Timeout status,
bit 1 - not used,
bit 2 - not used,
bit 3 - I/O error occurred,
bit 4 - Selected status,
bit 5 - Out of Paper Error,
bit 6 - Acknowledge status,
bit 7 - Not busy status.

Appendix G

INTERFACING TO DOS ENVIRONMENT

This appendix details some of the important functions that DOS provides for an application program.

PROGRAM SEGMENT PREFIX, ETC. When DOS hands over control of the CPU to a .EXE program, some of the registers are preset with data. In order for the program to run, it must preset the CS:IP registers to index the start of the application program execution logic. The SS:SP stack area is preset. The data values in DS and ES will be predefined to index the Program Segment Prefix (PSP). Inside the PSP, there are two areas of general importance: the environment pointer at location 2CH and the command line data string starting at location 80H. The byte at location 80H tells how many bytes of command line data follow starting at 81H. The data word at 2CH is a segment offset to index the start of the environment passed by the calling process. The environment data area contains ASCII strings of information like PATH=C:\. The environment can be seen at the DOS prompt by entering the SET command.

Example:

From a DOS command prompt, you enter the following line:

```
MODE CO80,43
```

DOS then tries to execute MODE.COM and passes the following ASCII data string in the PSP data area at DS:81H (note that the first character will be a space):

```
CO80,43
```

with the value 9 at 80H.

DOS CALLING Most DOS calls are made through the INT 21H function calls. With a standard DOS function call, you preset register AH with a DOS function code value and the other registers with the necessary data for the called function. Then you execute an INT 21H instruction to pass control of the CPU to DOS until it finishes the task and returns to your program. In general, if it returns with the carry bit set in the Flag register, then the function failed.

A main function of the DOS is to assist in handling files. When an application program calls the DOS to open or create a file, the DOS returns a file handle to the program. This is a 16 bit data word that the program needs to keep track of. When the application tries to access an open file, DOS will require the file handle to reference the correct file. The basic file functions are:

Create

On entry, AH=3CH, CX=file attributes, DS:DX index ASCIIZ string. The file attribute information is defined as:

bit 0 - file is read only if set,

bit 1 - hidden file if set,

bit 2 - system file if set,

bit 3 - volume label entry if set, only valid in root directory,

bit 4 - subdirectory entry if set.

On exit, if no carry, then AX=file handle data else if carry, then there is an error code in AX: 3 path not found, 4 too many files open, 5 access denied.

Open

On entry, AH=3DH, AL=access code, DS:DX index ASCIIZ string. The access code is defined as: 0 to open for reading, 1 to open for writing, 2 to open for both reading and writing.

On exit, if no carry then AX=file handle else if carry then there is an error code in AX: 2 file not found, 4 there are too many files open, 5 access denied, 12 invalid access code.

Write

On entry, AH=40H, BX=file handle, CX=count of number of bytes to write, DS:DX file buffer index.

On exit, if no carry then AX has count of writing else if carry then there is an error code in AX: 5 invalid handle, 6 access denied.

Read

On entry, AH=3FH, BX=file handle, CX=count, DS:DX=file buffer index.

On exit, if no carry then AX has count of number of bytes read else if carry then there is an error code in AX: 5 invalid handle, 6 access denied.

Close

On entry, AH=3EH, BX=file handle.

On exit, if no carry then function ok else if carry then there is an error code in AX: 6 invalid handle.

Seek

On entry, AH=42H, AL=move type, BX=file handle, CX:DX=move distance. Move type: 0 move from the start of the file, 1 move from the current file position, 2 move from the end of file. Move type 2 can be used to find the end of the file for appending data or to find the file size.

On exit, if no carry then DX:AX=new position else if carry then there is an error code in AX: 1 invalid function, 6 invalid handle.

Appendix H

DEBUGGING A COMPUTER PROGRAM

A good programmer should be able to find any software bug that exists in a computer program without the aid of a debugging software tool by analyzing the program listing. A good programmer will also note that a software debugging tool can sometimes speed up finding the location of a problem. When tracing down a bug, one of the primary objectives is to attempt to locate the defective segment of code as quickly as possible. Many times, this can be accomplished by just knowing what the user was doing with the program when the problem occurred. For instance, was the user saving a file, loading a new file, making changes to certain program features, etc.? Sometimes, this does not work because a bug can be initiated by one code segment but is not visible to the user until later on during program execution in another code segment. If a good examination of the current code listings does not produce a solution, then it may be time to use a software debugging tool.

One of the most common problems with Assembly language coding is indexing the wrong data cell with a pointer and letting a pointer move out of the defined data area for it. With the segment registers of the 80X86 language adding to addressing complexities, it is very easy to find yourself pointing to the wrong place.

The primary goal is to define and isolate the problem. Try searching the source code for key areas and write special debug code at key points. This will normally be just a message displayed to the standard output device that identifies the code segment that is being debugged. Then recompile and link the source with the debug code. Run the debug version and make note of the debug statements executed before the error condition occurs. You may have to repeat this process more than once to isolate and find a bad code segment, but you can usually narrow the search for a bad code segment very quickly with this method. The drawback with this method is that the code may have to be recompiled many times to find the problem. Sometimes, recompiling is very difficult and may take hours. If this is the case, then another method may be more useful.

There is a generalized logical approach to debugging almost any code. This simple method can be used to debug programs when you do not have access to the source code or can be used very efficiently with the source code. In almost all cases, a computer program will have subroutine calls of some kind or another. For this discussion, consider each subroutine call to be nested under the layer of the code that calls it. The concept and study of these layers can be a useful tool in debugging unknown code. In most systems, you can execute a debugger and have it load and make a program ready for execution. You should be able to single step through the program and take note of all the computer instructions executed. In this layered approach, you follow the top level of code until the error condition is noted. To follow the top level, simply single step through the code and test for the error condition after each instruction at the current layer. While single stepping through the code, each time a subroutine is called, skip over the instruction that calls or invokes the subroutine but make a note that the last instruction was a subroutine call and note the location of it. This means that you should not single step into the subroutine code but have the debugger execute it and return to the single step state when the subroutine is finished. At some point in the debugging process, the error condition will be noted. If the error condition is noted after a standard instruction, then the last instruction caused the

error condition and should be studied for a possible correction. If the last instruction was a subroutine call, then it is very likely that the subroutine was responsible for the error condition and should be investigated. Execute the program again up to the point where the subroutine procedure was called that caused the error condition to be noted. Then, single step into the subroutine and start searching this layer just like you did the last. You can then repeat the debugging process at this layer level the same way that you did in the top layer level until the error condition is totally isolated and the problem code is identified.

Sometimes, once a key subroutine is found, you may want to do a depth-first search on the subroutine instead of the breadth-first search described above. These search methods are not guaranteed to find all problems, but they provide a systematic way to find most program error conditions.

If the code is compiled in a high level language, you can normally associate Assembly language code statements to high level language code statements by the layer of the code level where they are located.

Appendix I

REAL TIME PROGRAMMING

Many people program in Assembly language to speed up program routines that take too much time in a high level language. All good high level language development systems should have an option to produce an Assembly language level source code listing of the compiled high level language source code. In most cases, this Assembly language level source code listing can be modified to improve performance of the software. This appendix discusses some methods commonly used.

One simple method is to check the code and remove all unnecessary NOP instructions. This should make the resulting code smaller and faster. Note that some NOP instructions may be needed by the code to execute correctly.

Another method is to look for unneeded instructions and remove them from the program code stream. Look for ways to take advantage of registers as storage area for temporary data.

When possible, replace all LEA instructions with MOV instructions using the OFFSET address option for the MOV. The MOV instruction is faster than the LEA.

Example:

```
lea     SI,data_string
```

changes to

```
mov     SI,offset data_string
```

Sometimes, you can convert multiply statements into bit shifts with adds to speed up the code. This method is discussed in the computer math section of this book.

When using a shift instruction with a shift count in register CL, the timing is very slow compared to a single bit shift for the standard 8086 processor. For the 8086 processor, the number of clock cycles required for a single bit shift in a register is two. The number of clock cycles required for a bit shift in a register with a shift count in register CL is 8 + (4 * CL). For the 80286 processor, the number of clock cycles required for a single bit shift in a register is two. The number of clock cycles required for a bit shift in a register with a shift count in register CL is 5 + CL. For the 80386 processor, the number of clock cycles required for a single bit shift in a register is three. The number of clock cycles required for a bit shift in a register with a shift count in register CL is three.

The XOR instruction can be used to set a register to zero faster than zero can be moved into the register with the MOV instruction.

The 80X86 processors use a prefetched instruction pipe to speed up execution. Program branching destroys the prefetched instruction pipe. Sometimes, reversing a jump condition and changing the code to match the reversed condition will increase speed.

Many language compilers offer the option of macro expansion which can be used in place of subroutine calls to speed up execution.

HARDWARE INTERRUPT TIMING CONSIDERATIONS

There are special timing problems that can occur when programming code is to be executed during a hardware interrupt. The primary concern is that the interrupt code must finish executing before the next interrupt from the hardware device occurs. To make sure the code can finish in time, the programmer may have to count clock timing cycles.

The following example demonstrates how these clock cycles add up. If the processor is running at 4.77 MHZ, then this translates to approximately 4,770,000 timing cycles per second. In this example, assume there is a hardware device that is interrupting at a rate of about 960 times per second (like a communications port running at 4800 BAUD with 480 receive interrupts per second and 480 transmit interrupts per second). For simplicity, round 960 up to 1,000 and divide this into 4,770,000 timing cycles per second. The result gives us about 4,770 timing cycles between each hardware interrupt at about 1,000 interrupts per second. Now if I say that the average instruction takes about 10 timing cycles, then I can say that you can only execute about 477 instructions between each hardware interrupt. If this hardware interrupt software routine requires the execution of more than 477 instructions per interrupt, you can assume that real time execution problems will occur. To be safe, the interrupt routine should allow for extra free cycles for the other hardware interrupting devices (such as the keyboard and the disk drives) to use.

If you tie into the system clock interrupt that ticks at a rate of about 18 times a second, you get the following figures:

(total cycles per second) / (18 ticks per second)

4,770,000 / 18 = 265,000	this is the number of timing cycles between each tick available

assume 12 cycles per instruction on average

265,000 / 12 = 22,083	this is the available number of instructions per tick

Some systems will use this clock tick interrupt to draw a mouse array item to the video screen. If the system is a 4.77MHZ PC and you find that each dot in the video array takes 100 instructions to update, then the maximum size of the video array will be limited to less than 220 dots or about a 10 by 20 dot array.

Appendix J

DIFFERENCES IN 80X86 PROCESSORS

A PC may have any one of these processors to be classified as an IBM PC compatible: 8088, 8086, 80188, 80186, 80286, 80386, 80486, 80X86. From a general hardware point of view, the main difference between these processors is the size of the data bus, the size of the address bus, and the clock speed of the processor. From a software programmer's point of view, the processors are very similar. The 8088 and the 8086 have the same instruction set. The 80188 and the 80186 have the same instruction set which includes all the 8086 instructions plus a few additional instructions. The 80286 executes all the same instructions that the 80186 executes plus more. The 80386 executes all the same instructions that the 80286 executes plus more. The 80486 executes all the same instructions that the 80386 executes plus more. (There are some exceptions to the backward compatible rules which a software programmer can use to identify which of the different processors the software is on.) This appendix discusses the primary differences between the processors from a general software application point of view.

ADDITIONAL 80186, 80188 INSTRUCTIONS The PUSHA instruction to push all registers onto the stack along with the POPA instruction to pop all registers from the stack. An instruction to push an immediate data word onto the stack.

An instruction to enter a procedure based on a compiler stack construct along with an instruction to leave a procedure based on a compiler stack construct. Instructions to shift and rotate using an immediate data value instead of a data value in the CX register. The BOUND instruction for the testing of a data value against an upper and a lower limit. An instruction for string input of port data along with an instruction for string output of port data.

ADDITIONAL 80286 INSTRUCTIONS The 80286 offers a protected supervisor mode with associated special jumps, calls, and supervisor registers to support it. An interrupt instruction with immediate data. The 80286 has expanded memory addressing which is controlled with additional supervisor instructions and registers.

ADDITIONAL 80386 INSTRUCTIONS The 80386 is a 32 bit processor with a backward compatible 16 bit mode. All the standard registers are expanded to 32 bit registers. There is also the addition of two new data segment registers: FS and GS. The instruction set is expanded to include support for all the new 32 bit registers. To reference a 32 bit register, put an E in front of the 16 bit reference. For example, EAX is the 32 bit register reference for the 16 bit register AX. The conditional jumps were expanded from eight bit offsets to 16 bit offsets. A set of new bit handling instructions was included. A new instruction to set a byte value to zero or one depending on the current condition codes was included. There is a backward compatible 8086 mode that the 80386 provides. The virtual memory management hardware provided can access a very large address space depending on which particular version of the 80386 chip you are using.

ADDITIONAL 80486 INSTRUCTIONS The XADD instruction to exchange and add is new. This instruction moves the destination operand into the source location and adds the source operand to the destination operand, putting the result in the destination. The CMPXCHG instruction to compare and exchange is new. This instruction compares the accumulator to the destination operand. If they are equal, the source operand is loaded into the destination. If they are not equal, the destination is loaded into the accumulator. There are

several system supervisor level instructions including cache control instructions, etc. The 80486DX chip has the 8087 processor functions included inside in the chip hardware. The 80486SX chip does not have the 8087 processor functions included inside the chip hardware.

Appendix K

KEYBOARD CODE TABLE

This is a table of common keyboard code values that are returned to a program when it requests keyboard input data. If this is a BIOS call, the values are returned to the program in register AL unless the value of AL is zero. The values with 0: are extended code values with data in AH. If this is a DOS call and you receive a zero value, you must make another DOS call to get the extended code value.

Key	*Code*	*SHIFT code*	*CTRL code*	*ALT code*
A	97	65	1	0:30
B	98	66	2	0:48
C	99	67	3	0:46
D	100	68	4	0:32
E	101	69	5	0:18
F	102	70	6	0:33
G	103	71	7	0:34
H	104	72	8	0:35
I	105	73	9	0:23
J	106	74	10	0:36
K	107	75	11	0:37
L	108	76	12	0:38
M	109	77	13	0:50
N	110	78	14	0:49
O	111	79	15	0:24
P	112	80	16	0:25

Key	*Code*	*SHIFT code*	*CTRL code*	*ALT code*
Q	113	81	17	0:16
R	114	82	18	0:19
S	115	83	19	0:31
T	116	84	20	0:20
U	117	85	21	0:22
V	118	86	22	0:47
W	119	87	23	0:17
X	120	88	24	0:45
Y	121	89	25	0:21
Z	122	90	26	0:44
1	49	33		
2	50	64	0	
3	51	35		
4	52	36		
5	53	37		
6	54	94	30	
7	55	38		
8	56	42		
9	57	40		
0	48	41		
-	45	95	31	
=	61	43		
[	91	123	27	
]	93	125	29	
\	92	124	28	
;	59	58		
'	39	34		
,	44	60		
.	46	62		
/	47	63		
`	96	126		

Key	*Code*	*SHIFT code*	*CTRL code*	*ALT code*
F1	0:59	0:84	0:94	0:104
F2	0:60	0:85	0:95	0:105
F3	0:61	0:86	0:96	0:106
F4	0:62	0:87	0:97	0:107
F5	0:63	0:88	0:98	0:108
F6	0:64	0:89	0:99	0:109
F7	0:65	0:90	0:100	0:110
F8	0:66	0:91	0:101	0:111
F9	0:67	0:92	0:102	0:112
F10	0:68	0:93	0:103	0:113
F11	0:133	0:135	0:137	0:139
F12	0:134	0:136	0:138	0:140

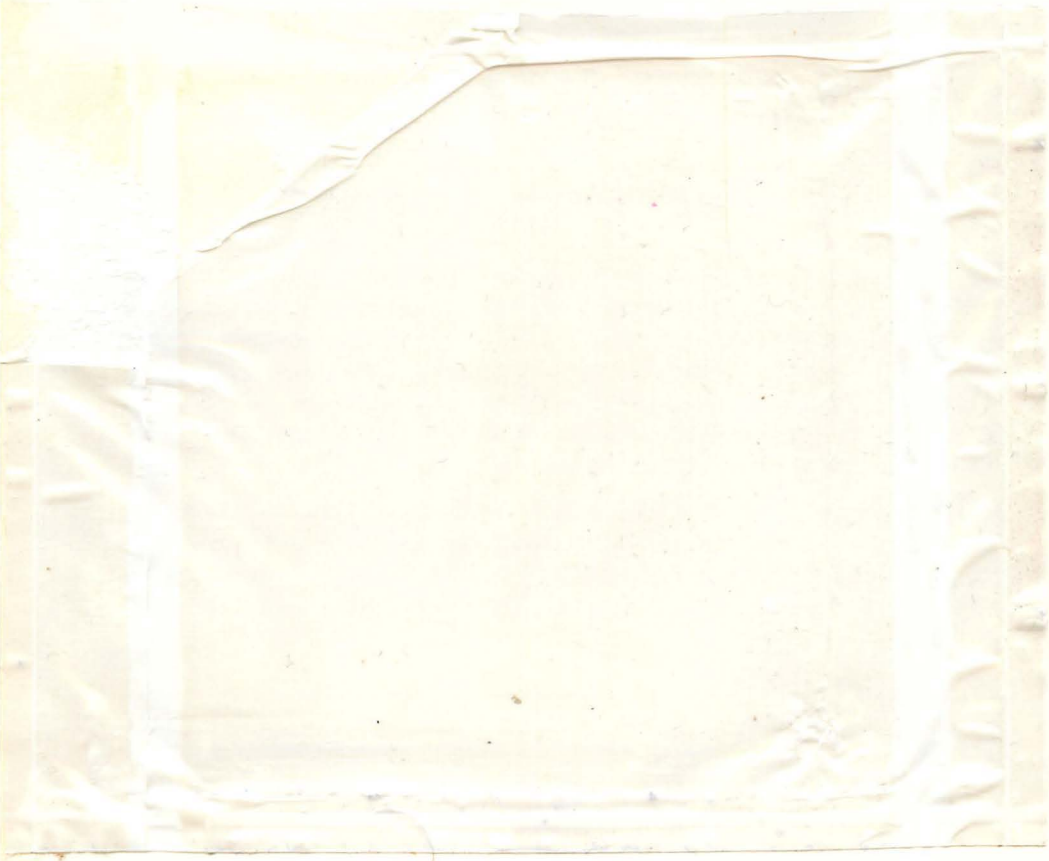
Home	0:71
Up Arrow	0:72
Page Up	0:73
Left Arrow	0:75
Right Arrow	0:77
End	0:79
Down Arrow	0:80
Page Down	0:81
Insert	0:82
Delete	0:83

INDEX

273
27833-

Gail
8815 N.E. 74th St.
KC. Mo., 64158

WROX Press Ltd.
feedback@ wrox.demon.co.uk
312-465-3559
WROX Press Ltd.
2710 W. Touhy
Chicago, IL 60645



Master Class Assembly Language
WROX
$49.95
649-711
Contains 63 pgs VIRUS and ANTIVIRUS

Revolutionary Guide to Assembly Language
WROX
$44.95

Revolutionary Guide to Bit Mapped Graphics

POPULAR APPLICATIONS SERIES

LEARN Turbo Assembler IN A DAY

FOR THE IBM®-PC, PS/2®AND COMPATIBLES

Develop a good foundation in assembly language programming with this book as your guide. Designed for the beginner, this book teaches users of Borland's Turbo C and C++ family of products how to program using the bundled Turbo Assembler language compiler. From programming structure to the creation of running applications, learning how to use this powerful language has never been easier. Companion diskette with source code examples included.

STEPHEN K. CUNNINGHAM is a professional programmer and consultant with an extensive background in the design and programming of computer applications. Mr. Cunningham's published works include numerous software manuals for a variety of computer applications.

Book Buyer's Guide

User Category: ☑ New ☑ Experienced ☐ Advanced ☐ Professional ☑ Educator

Book Type: ☑ Tutorial — for: ☑ Self Teaching ☑ Classroom Use
☐ Command Reference
☐ Advanced Topics
☑ Working Examples — ☑ Source Code ☐ Templates ☑ Hands-on Activities ☐ Other
☑ Companion Diskette — ☑ Included ☐ Available

9 781556 223006

ISBN 1-55622-300-5

Cover Design and Photo by Alan McCuller

WORDWARE PUBLISHING, INC.
—First in Quality—

$15.95